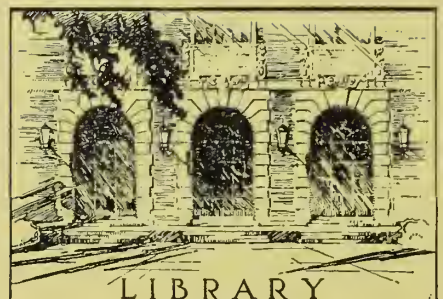


LIBRARY
OF THE
UNIVERSITY
OF ILLINOIS

510.84
Il6r
no.237-242
cop.2

The person charging this material is responsible for its return on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

University of Illinois Library

MAR 18 1971
MAR 22 RECD
DEC 7 1972
NOV 9 RECD

L161—O-1096

Digitized by the Internet Archive
in 2013

http://archive.org/details/translatorgenera237mcin

IRLib
no. 237
cop. 2

Report No. 237

COO-1018-1121

231-242

THE TRANSLATOR GENERATOR SYSTEM

by

A. W. McInnes

July 12, 1967

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

THE LIBRARY OF THE
NOV 6 1967
UNIVERSITY OF ILLINOIS

Report No. 237

THE TRANSLATOR GENERATOR SYSTEM

by

A. W. McInnes

July 12, 1967

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Supported in part by Contract AT(11-1)-1018 with the U.S. Atomic Energy Commission and the Advanced Research Projects Agency.

237-24

THE TRANSLATOR GENERATOR SYSTEM

A. W. McInnes

1. INTRODUCTION

This paper describes a system designed to produce a scanner from the Backus Naur Form definition of the syntax of a programming language. The theoretical aspects of this problem are considered in the paper by Eickel and Paul [1].

This problem really involves two important problems:

(a) Parsing problem

To give a method which decides whether or not a given string, according to some rules of a language, belongs to it, and if so to give the corresponding rules.

(b) Ambiguity problem

To decide whether or not a language is unique, i.e., whether or not for every string in the language the corresponding set of rules is uniquely determined.

In this paper we restrict our attention to Chomsky two grammars (i.e., having a production system having just one element of the alphabet of the language as left hand side of each production) having just one distinguished element, Z, of the alphabet to which every sentence in the language may be reduced by repeated applications of the members of the production system.

The main purpose of this system is to produce a purposive parsing system i.e., if in a left to right scan of a string, we find any production (replacement rule in the syntactical definition of the language) matched by a substring of the given string, we can make the indicated replacement knowing that this is the only possible replacement. However, due to the practical necessity of providing bounds on the algorithm at various stages, it is not clear that such a goal is possible for every

such language considered. The size of the subset which can be successfully treated will be a matter of practical experimentation, and possible modification based on experience with the system.

1.1 Basic Concepts

Let $\mathcal{A}$ = {A, B, C, ..., Z} be a finite set of distinguished characters which we shall call the alphabet. The elements (a, b, c, ...) of the free semigroup $\mathcal{F}(\mathcal{A})$ over $\mathcal{A}$ under concatenation will be called strings over $\mathcal{A}$. The unit element (or empty string) of $\mathcal{F}(\mathcal{A})$ will be denoted by ε. If $x = X_1 X_2 \ldots X_n$ ($X_k \in \mathcal{A}$), then $|x| = n$ is called the length of x. A finite set Π of ordered pairs of strings,

$$\Pi = \{ a_k :: = b_k \; ; \; a_k, b_k \in \mathcal{F}(\mathcal{A}) \sim \{\varepsilon\} \; ; \; k = 1, 2, \ldots, m \}$$

is called a production system in $\mathcal{F}(\mathcal{A})$. A production $a_k :: = b_k$ (read b_k may be replaced by, or becomes, a_k') may be regarded as a replacement rule in the defining syntax of the language. a_k is called the left side and b_k the right side of the production $a_k :: = b_k$. Π generates relations ρ_o and ρ in $\mathcal{F}(\mathcal{A})$ in the following manner:

Definition 1.

$$u, v \in \mathcal{F}(\mathcal{A}) \; : \; u \rho_o v \text{ iff } \quad x, y \in \mathcal{F}(\mathcal{A}) ,$$

$$\exists \; (a :: = b) \in \Pi \ni u = xby, \; v = xay$$

i.e., for given strings u and v in the free semigroup, we define u in relation to v ($u \rho_o v$) if and only if there exist strings x and y in the free semigroup and a production a :: = b in the production system Π, such that the string u has the form x b y and v has the form x a y.

Definition 2.

$$u, v \in \mathcal{F}(\mathcal{A}) \; : \; u \rho v \text{ iff } \exists \{u_k \in \mathcal{F}(\mathcal{A}) \} \; k=0,1,2 \ldots, n \ni$$
$$u_o = u, \; u_n = v, \; u_k \rho_o u_{k+1} \quad (k = 0,1,2, \ldots, n-1)$$

i.e., we define the string u to be in relation to the string v (u ρ v) if and only if there exists a finite sequence of strings u_k such that $u_k \rho_o u_{k+1}$ according to definition 1, and such that we are led by repeated applications of the relation ρ_o from the string u to the string v. That is the relation ρ is the transitive closure of the relation ρ_o. For u ρ v we shall say that u is a word for v.

A special subset of $\mathcal{A}$ is distinguished:

$\mathcal{A}^{\perp}$ = { X | X ε $\mathcal{A}$, X does not occur as a constituent of the right side of a production of Π }

We call the subset $\mathcal{A}^{\perp}$ the set of anti-terminal characters in the alphabet.

The system we wish to consider is a quadruple $\Sigma = (\mathcal{A}, \Pi, Z, \gamma)$ where $\mathcal{A}$ is an alphabet, Π is a production system in $\mathcal{F}(\mathcal{A})$, Z is the distinguished element in $\mathcal{A}^{\perp}$, and

$$\gamma = \{ x \mid x \varepsilon \mathcal{F}(\mathcal{A}) , x \underline{\rho} Z \}$$

i.e., the set of all strings in the free semigroup which are words for the distinguished element Z, including the case x = Z. However note that we have restricted our consideration to those systems having a production system of which the left side of each production is just a single element of the alphabet $\mathcal{A}$. The set γ of such strings is called a language.

Definition 3 (critical production):

A production $A :: = B_1 B_2 \ldots B_m$ is called critical if there exists a $k \varepsilon \{ 1,2, \ldots m\}$ and a production $A' :: = b'$ in Π, not identical with the production given above such that B_k occurs the string b'.

Using this concept we now define a relation $\bar{\rho}_o$ contained in ρ_o.

Definition 4

$$u, v \in \mathcal{S}(\mathfrak{A}): u \,\bar{\rho}_o\, v \quad \text{iff} \quad x, y \in \mathcal{S}(\mathfrak{A}),$$

$$\exists\ (a ::= b) \in \Pi \ni a ::= b \text{ is not critical},$$

$$u = xby \quad , \quad v = xay.$$

i.e., the relation $\bar{\rho}_o$ is just the relation ρ_o restricted to the non-critical productions. Corresponding to definition 2, we also define the relation $\bar{\rho}$ as the transitive closure of $\bar{\rho}_o$; that is $u \,\bar{\rho}\, v$ if we can pass from u to v by a finite sequence of applications of $\bar{\rho}_o$ using only the non-critical productions. Clearly ρ contains $\bar{\rho}$.

The relation $\bar{\rho}$ has the following property:

$$(\#) \quad u, v, w \in \mathcal{S}(\mathfrak{A}): \quad u \,\bar{\rho}\, v,\ u \,\rho\, w \Rightarrow v \,\underline{\underline{\rho}}\, w$$

To demonstrate this property by an example let us suppose we have productions:

$$A_1 ::= B_1$$

$$A_2 ::= B$$

$$A_3 ::= B$$

The first is the only non-critical production in this system.

Let us suppose $u = a\ B_1\ b\ B\ c\ B\ d$

Thus if $v = a\ A_1\ b\ B\ c\ B\ d$

We have $u \,\bar{\rho}\, v$

Now if $w = a\ A_1\ b\ A_2\ c\ A_3\ d$

then $u \,\rho\, w$ since for ρ we may use any of the productions and we have the following sequence of relations ρ_o:

$u = a\ B_1\ b\ B\ c\ B\ d\ \rho_o\ a\ B_1\ b\ B\ c\ A_3\ d\ \rho_o\ a\ A_1\ b\ B\ c\ A_3\ d\ \rho_o\ a\ A_1\ b\ A_2\ c\ A_3\ d$

and note that these two relations satisfy the condition that $\bar{\rho}$ is just the relation ρ restricted to the non-critical productions.

Thus we clearly have $v\ \rho\ w$ since

$v = a\ A_1\ b\ B\ c\ B\ d\ \rho\ a\ A_1\ b\ B\ c\ A_3\ d\ \rho\ a\ A_1\ b\ A_2\ c\ A_3\ d$

Note that this property # is not true for ρ since we could then have

$u\ \rho\ v_1 = a\ A_1\ b\ A_3\ c\ B\ d$ and

$u\ \rho\ w = a\ A_1\ b\ A_2\ c\ A_3\ d$.

In this case it is not true that $v\ \underline{\rho}\ w$ since although

$v_1\ \rho\ v_2 = a\ A_1\ b\ A_3\ c\ A_3\ d$,

there is no way we can replace the element A_3 between the substrings b and c to become the element A_2 required for the string w. Thus we have come to a "dead end" and have been led outside the set γ consisting of the strings comprising our language.

Thus if the relation ρ_o (ρ) had this property (#), then the parsing problem would become trivial for then

$u \in \gamma,\ v \in \mathfrak{F}(\sigma) : u\ \rho_o\ v \Rightarrow v \in \gamma$,

and this would mean that the application of the syntactical rules according to ρ_o never leads from inside to outside of γ. In general however, ρ does not have this property. Therefore, the main goal to be achieved is to restrict the production system Π to a syster by means of contextual conditions such that the orrespondingl stricted relation ρ^* ($\subset \rho$) has the above mentioned property (#).

All ambiguities in the language are characterized by the fact that they are words for Z according to more than one essentially different ways. In other words we may obtain more than one distinct sequence of

replacement rules (or productions) which may be applied to a given string and reduce it to the distinguished element Z. Thus the given string may be regarded as a valid word in the language in more than one essentially different ways - i.e., in the language of programming, it may be scanned in more than one way and still turn up as an apparently legal construct. This is essentially the problem as discussed above except that the "wrong choice" of a production now leads again into the set γ instead of outside of γ and to a "dead end" as before.

In order to rigorously explain "essentially different" as used in the previous paragraph we give the following definitions:

Definition 5 (local ambiguity):

A string x is called locally ambiguous if there exist in Π two different productions $a :: = b$ and $a' :: = b'$ with $|x| < |b| + |b'|$ where b and b' are substrings of x, which cover x completely.

Thus since the length of the string x is less than the sum of the lengths of the strings b and b' and since $\{b\} \cup \{b'\}$ covers x completely, it follows that $\{b\} \cap \{b'\} \neq \phi$. Thus it is clear that the strings b and b' have at least one element in common and so since the productions are different, if there exist two such productions they must be critical by definition 3. From definition 5 it follows that for every locally ambiguous string x, there exist u, v, u', v' ε $\mathcal{V}(\sigma)$ with

$$x = u\ b\ v = u'\ b'\ v' \quad ,$$

$$x\ \rho_o\ u\ a\ v = x_1$$

$$x\ \rho_o\ u'\ a'\ v' = x_2$$

Definition 6 (ambiguity):

A string m is called ambiguous iff there exist u, v ε $\mathcal{V}(\sigma)$ and a locally ambiguous string x (where $x\ \rho_o\ x_1$, $x\ \rho_o\ x_2$) such that

$$m \underline{\rho} u\ x\ v \quad , \quad u\ x_1\ v \underline{\rho} Z \text{ and } u\ x_2\ v \underline{\rho} Z$$

This rigorously defines what we mean by saying that the string m is a word for Z in two essentially different ways, and, places in terms of the concepts rigorously defined above, our intuitive idea of an ambiguity as a construct having two (or more) apparently valid interpretations within the structure of the language.

Thus it is clear, that in order to isolate those elements of Π which are causing either ambiguities occuring in Σ, or the relation ρ not having the quoted property(#), we need to isolate the critical productions of Π. We then attempt to resolve these problems by finding contextual conditions on the critical productions and obtain a unique system which contains no ambiguities.

The remainder of this report gives details of how this goal may be achieved. Chapter 2 deals with the reduction of our system to a Simple Chomsky system; Chapter 3 considers the determination of the contextual conditions for the critical productions and Chapter 4 describes a purposive scanner obtained from the system.

1.2 Example

In order to better explain some of the concepts mentioned above, we shall illustrate by means of a short example.

Suppose we have the production system:

F :: = V
F :: = (E)
T :: = F
T :: = T * F
E :: = T
E :: = E + T
Z :: = E

It will be seen that these are part of the productions for an arithmetic expression in ALGOL, where V = variable, F = factor, T = term and E = expression, which becomes the distinguished element in the system.

Suppose we consider the string V + V * V. Note that all productions except the first are critical so we may expect that some methods of reducing this string will lead to dead ends.

There is a production F :: = V. So suppose we replace all the elements V in the string by F. The string now becomes

F + F * F

There is a production T :: = F, so we can replace every F by a T. The string becomes

T + T * T

Similarly, using the production E :: = T, the string becomes

E + E * E

and finally using the production Z :: = E, the string becomes

Z + Z * Z.

However, we have now reached a dead end since the string has not been reduced to the single distinguished element Z and no further reductions form the production system can be made. Let us consider what went wrong. By an inspection of the production system and our intuitive knowledge in this case, we find that the problems began in replacing the F by T. If the final F in the string had not been replaced, we would have obtained the string

T + T * F

and the last three elements of the string is a right side of a production, T :: = T * F. Making this replacement we obtain the string

T + T

However, if we again use the production E :: = T in both cases here, we are again led to a dead end

Z + Z

But if we use this production to replace only the first T we obtain the string

E + T

which is again a right side of a production E :: = E + T.

Making this replacement and then using Z :: = E, we have finally reduced the string to the single element Z as required

Thus, our reduction of the original string V + V * V, would have been satisfactory if we use the production T :: = F to replace F by T, except when F is preceded by a *. Or expressing the same condition constructively, we use the production T :: = F only when F is preceded by a '(' or a '+' or 'ε' (ε is the null string). Thus we replace the production T :: = F in the system by the three productions:

εT :: = εF

(T :: = (F

+T :: = +F

Then we can make purposive replacements of F, using any production whose right side matches a substring of the given string, and then being sure that we will not be led to a dead end.

Similarly for the production E :: = T, we find that we may replace T by E except when T is preceded by a '+'. Hence we replace this production in the system by the three productions

$$\varepsilon E :: = \varepsilon T$$

$$(E :: = (T$$

$$*E :: = *T$$

We now have a purposive passing scheme using this modified production system which avoids leading to dead ends by the addition of contextual conditions to certain productions. The following system attempts to implement in a systematic manner, our intuitive evaluation of this simple system.

2. SIMPLE CHOMSKY SYSTEMS

The first part of this system will transform the production system of a CHOMSKY - language given in Backus Naur Form into a simple production system for the same CHOMSKY - language. The essential property of a simple production system is that its critical productions are easily recognizable and are all of length one. The description of the implementation given in section 2.2 should be read in conjunction with the flow charts for the first part given in the appendix.

2.1 Basic Concepts

We are given a CHOMSKY system, which includes a set of productions Π. Initially we reduce the given CHOMSKY system to a simple CHOMSKY system which implies that the set of productions Π must be transformed to a set Π' where Π' is defined in definition 7.

Definition 7 (Simple CHOMSKY System)

A CHOMSKY System $\Sigma' = \{ \mathcal{A}, \Pi', Z, \gamma' \}$ is called Simple if

$\Pi' = \{ A_k :: = b_k \ ; \ k = 1, 2, \ldots n \}$

has the following properties:

1) $|b_k| \leq 2$ i.e., the right sides of the productions have length one or two. Thus all productions of Π' have the form:

$$A_i :: = S_i \, T_i \qquad i = 1, 2, \ldots\ldots, \ell$$

$$A_{\ell+j} :: = U_j \qquad j = 1, 2, \ldots\ldots, m$$

2) The sets $\{S_i\}$, $\{T_i\}$, $\{U_j\}$ are pairwise disjoint.

3) $S_p \neq S_q$, $T_p \neq T_q$ for all $p, q = 1, 2, \ldots, \ell$ and $p \neq q$.

Thus all the critical productions are of length one and the local ambiguities (i.e., having more than one replacement possible from a given element) are exactly the right-hand sides of the critical productions.

The simple CHOMSKY system is obtained from the given system by carrying out the following operations:

i) If there is a production $X :: = X_1X_2...X_n$ where $n > 2$ then a new character E is created and this production is replaced by the pair of productions:

$$E :: = X_1X_2$$

$$X :: = E\ X_3 \ldots\ X_n$$

The second of these new productions is of length $n - 1$. If $n - 1 > 2$ then this procedure is repeated iteratively until all the new productions have length 2. The new character E is adjoined to the alphabet $\mathcal{A}$.

ii) If there exist two productions of the form
A :: = XT; B :: = SX (or a production of the form A :: = XX), then the symbols X_L, X_R are created and the productions are replaced by

$A :: = X_LT$	or	$A :: = X_L\ X_R$
$B :: = SX_R$		$X_R :: = X$
$X_R :: = X$		$X_L :: = X$
$X_L :: = X$		

The new characters X_L, X_R are adjoined to the alphabet $\mathcal{A}$.

iii) If there exist two productions of the form
A :: = XT (A :: = SX) and B :: = X, then the symbol E is created and the first production is replaced by
A :: = ET (A :: = SE)
E :: = X

The new character E is adjoined to the alphabet $\mathcal{A}$.

iv) If there exist two productions of the form

$A :: = ST$ and $B :: = ST$

then a new symbol H is created and these productions are replaced by

$H :: = ST$

$A :: = H$

$B :: = H$

The new character H is adjoined to the alphabet $\mathcal{A}$.

v) If there exist two productions of the form

$A :: = XT_1$ and $B :: = XT_2$ where $T_1 \neq T_2$

(or $A :: = S_1X$ and $B :: = S_2X$ where $S_1 \neq S_2$)

then the symbols G, H are created and these productions are replaced by

$A :: = GT_1$ $\qquad$ $(A :: = S_1G)$

$G :: = X$

$B :: = HT_2$ $\qquad$ $(B :: = S_2H)$

$H :: = X$

The new characters G, H are adjoined to the alphabet $\mathcal{A}$. This construction, if carried out in the given order, leads after finitely many steps to a simple CHOMSKY system. The proof of this assertion is given in the paper by Eickel and Paul [1].

2.2 Implementation

2.2.1 Functional Description

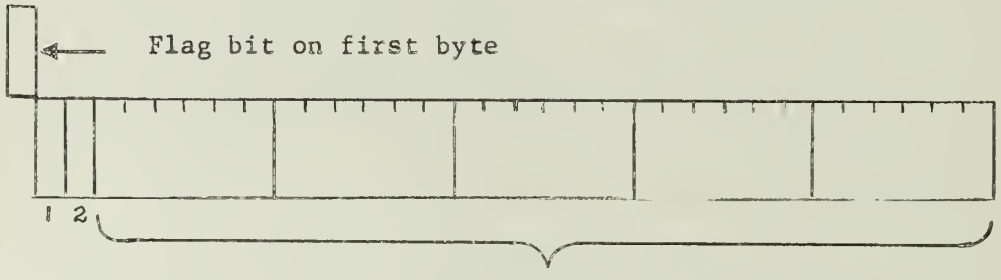
In order to accomplish this reduction and also in order to facilitate the next stage in which the ambiguous productions are resolved by means of context, the productions Π' are collected in three stacks:

(1) Stack S_2 (with stack pointer p_2) which contains all productions of length 2.

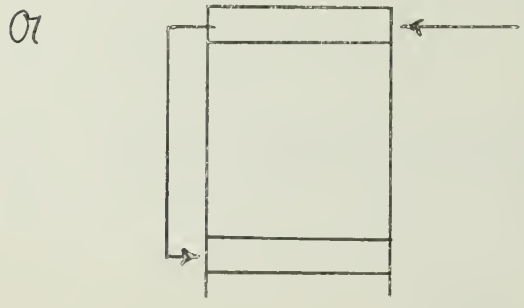
(2) Stack S_1 (with stack pointer p_1) which contains all unambiguous productions of length 1.

(3) Stack S_u (with stack pointer p_u) which contains all critical productions (of length 1 by hypothesis).

As each production of the given system is read in, each new alphabet element is entered in the symbol table $\mathcal{A}$ in the first available location. The location of this element in the table $\mathcal{A}$ gives its internal number which is used in all subsequent manipulations. Each element of the symbol table $\mathcal{A}$ is of one word in length and has the following form:



Five alphabetic characters of external representation.



If the new element is an anti-terminal symbol (i.e., a member of σ^1 which consists of all those symbols that do not appear on the right-hand side of a production), then the flag bit on the first byte is set to one. The first two bits of the word are used to signal certain conditions on the symbol represented in the rest of the word. If both bits are zero then the external symbol is not more than five alphabetic characters in length and is represented in BCD code in the remaining 30 bits of the word. If both bits are one then the external symbol is more than five alphabetic characters. In this case only the first character is represented in BCD code in the remaining six bits of the first byte and the last three bytes contain the link address of the first byte of the sequence of consecutive bytes which contain the remaining characters of the extended symbol. The flag bit is set on the last byte of this sequence to indicate the length of the extended symbol. If the first bit is one and the second bit is zero then a "created symbol" is indicated. In this case the external representation is left blank, but the internal number corresponding to this position is used in the productions to represent a created symbol as outlined above. To summarize:

Bit 1	Bit 2	
0	0	External symbol of length $\leq$ 5 alphabetic characters
1	0	Created symbol
1	1	Extended external symbol i.e., length $>$ 5 characters.

In order to conserve storage and yet not be limited by the number of symbols that may be used in a given system, the internal number will be represented in the following form:

Since in general the reduced system will not have more than 255 symbols in its "alphabet", the internal number of each system will be represented by one byte. If however there are more than 255 symbols in the alphabet then one byte is added to allow another 255 elements to be

represented. This process is continued for as large an internal number as required i.e., if the first byte has the value 255 (all bits are units) then to find the internal number the value of the next byte is added to 255. If the second byte also has the value 255 then the next byte is considered and its value is also added on.

e.g.

11111111	11111111	00000011

Internal number = 255 + 255 + 3

The forms of the stacks S_2, S_1, and S_u are

S_2:

a_i	S_i	T_i

$1 \leq i \leq p_2 - 1$

S_1:

b_j	U_j

$1 \leq j \leq p_1 - 1$

S_u:

c_k	V_k

$1 \leq k \leq p_u - 1$

where the symbols denote the i-th, j-th, k-th elements of the respective stacks, and where each symbol represents an internal number of one byte in length in general, but extended as above when necessary.

In the procedure outlined below for making the required reduction to a simple Chomsky system, the operations described previously are combined in a series of subroutines. As each production is read in, each symbol is compared with the symbols already in the symbol table $\mathcal{A}$, and converted to its internal number. The production, represented in internal number form is thus built up in a special location called PRØDN, from which it is pushed into the appropriate stack after any necessary operations have been carried out. If these operations involve the creation of a new symbol, then the 'created symbol' bit is set on in the next available cell in the symbol table $\mathcal{A}$, the external symbol representation is left blank, and the corresponding internal number is used in PRØDN as the created symbol. The pointer p_a is of course increased by one. The symbols of PRØDN are denoted by a, S, T (not subscripted) for the zeroth, first and second symbols respectively. The procedure to accomplish the reduction and set up the system for the resolution of ambiguities, is a relatively simple one which calls on three subroutines: CØMPAR, E and RHØ which will be explained in detail below. Note that the left hand side of a production is called the zeroth element and the first, second etc. elements of the right hand side are termed the first, second etc. elements respectively of the production. The parameter n determines the position of the symbol of the production currently under consideration and the parameter j is a measure of the number of elements of the right-hand side of the productions which match previous right hand sides (see Subroutine E(n)).

2.2.2 Operation Description

The main procedure SIMPLE

A new production is read in, the location PRØDN (where the production is to be assembled in its internal form) is cleared, the parameters n, j are initialized and a call is made on the subroutine CØMPAR to find if this symbol has already been used in a production. The next symbol is tested and if this is empty and if n = 0 (i.e., a left hand side was being considered) then there is an error condition, ERRØR 1, since the production did not have a right hand side. If n $\neq$ 0 then, since all symbols of this production have been read in, a call is made to the subroutine E which will push the production in the appropriate stack depending on the value of the parameter j. If, however, the next symbol is not empty, then if it is not the third symbol, the comparison procedure with the symbol table $\mathcal{A}$ is again repeated with this new symbol by means of the subroutine CØMPAR, as before. If the next symbol is the third then the subroutine RHØ is called, which separates off a production of length two as described in operation (i) above and third symbol now becomes the second before repeating this process of comparison. Eventually the "next symbol" will be empty (the productions are of finite length) and the procedure loops back, as above, to consider the next production.

Subroutine CØMPAR (n, j)

This procedure searches through the symbol table $\mathcal{A}$ (whose i-th element is denoted by N_i) to check if the n-th symbol of the production matches any element of $\mathcal{A}$, i.e., the n-th symbol has been used in a previous production. If no match is found then the new symbol is entered into the symbol table $\mathcal{A}$. Note that if the symbol is longer than five alphabetic characters then only the first character is entered into $\mathcal{A}$ and the remaining bytes contain a link address to the extension as explained previously. The pointer p_α is the internal number of this new symbol and this is placed in

the n-th position of PRØDN (where if the internal number is greater than 255 it is converted to the representation described above). If n = 0 the symbol is a left hand side and so the flag bit is initially set, to denote that this symbol possibly is an anti-terminal symbol.

However if a match is found and we are not considering a left-hand side (i.e., n ≠ 0) then a check is made to determine if the matched symbol was provisionally marked as an anti- terminal symbol (i.e., the flag bit is set). If this is the case then the flag bit is set to zero since we are considering a right hand side and the symbol, therefore, is not anti-terminal. If the flag is zero then this right-hand side symbol matches a right-hand side symbol of a previously considered production and the parameter j is adjusted accordingly. Whether the symbol is a right or left hand side symbol the loop variable i (converted to the required representation if necessary) is placed in the n-th position of PRØDN and a return is made to the calling point of the procedure. Note if n = 1 and this first element of the right-hand side was not matchedchen j is set to the value 1/2 before the return, in order that j has the correct value consistent with the description in the subroutine E(n) in the case that the second element is matched.

Subroutine RHØ (n)

This procedure separates off a production of length two from a production of length greater than two, as described in operation (i). A new symbol is created in the manner previously described. Since the zeroth element (left hand side) of the production will be needed for further processing of the over length production, this is stored temporarily before placing the newly created symbol (in internal number form) in the zeroth position. Then, since the subroutine E must be called to determine into which stack this production of length two is to be pushed (since there may

have been matches in the first two elements, the subroutine E will destroy the contents of PRØDN) the contents of PRØDN are stored temporarily before calling E. Upon return from E, processing of this production is continued with the original left hand side as left hand side of the production and the left-hand side of the split off production as the new first element. The third symbol of the production being processed now becomes the second symbol upon return. Note that in the main procedure, after calling RHØ, the parameter j is set to one half. This is because the first symbol is now the created symbol and so cannot match any symbol of a previous production.

Subroutine E(n)

This is the main subroutine which determines into which stack the production is to be pushed. It combines all the operations (ii), (iii), (iv), (v) in case any of these conditions are met. This subroutine in turn calls on three further subroutines:

a) REPL4 in case condition (iv) is met,
b) REPL2 in case either conditions (ii), (iii) or (v) are met,
c) SS1 which searches the stacks of productions of length one in case the matching element was not found among the elements of S_2.

The parameter n is the measure of the number of matching right-hand sides. If the parameter n = 0 then no matches were found and so if the second element T of PRØDN is empty then the production is pushed into S_1 else it is pushed into S_2 before making a return.

If the parameter n = 1 then the first element of PRØDN was matched but not the second. If n = 2 then the second element of PRØDN was matched but not the first. If n = 3 both elements of production were matched. Thus if $n \neq 0$ the first and second elements of S_2 are searched from the top down

to attempt to find where the matching element occurs. If no match has been found when the bottom of S_2 has been reached then before the subroutine SS1 is called to search S_1 and S_u for the match, we check whether n = 2, and if so, whether the matched element in fact matches the first element of the production under consideration. In this case we call the subroutine REPL2 to effect the replacements called for by condition (ii). We set the second parameter of REPL2 to be the T element of this new production which will be pushed into S_2 by the subroutine REPL3 which is called by REPL2. If n = 2 the second element of PRØDN was matched and so SS1 is called with parameter T (the second element of PRØDN). If n = 1, the first element was matched and so SS1 is called with parameter S. However if n = 3, after calling SS1 (S) and determining the match for the first element, the match for the second element has yet to be determined and so the production must be popped from S_2 and the same production evaluated again for the match of the second element by calling SS1 (T).

On the other hand if a match is found in S_2, the resulting action again depends on the value of the parameter n. (if n = 2 only T need be checked against the element of S_2). If T matches an element of S_2 then subroutine REPL2 with appropriate parameters, is called which essentially makes the replacements outlined in operations (ii) and (v). Then if $n \neq 3$ a return is made, otherwise, the match of the first element S has yet to be found and so the production is popped from S_2 for further consideration.

The parameter n is set to 1 (first element matched) and the search of S_2 is continued since this match may also be in S, but among the elements as yet unsearched.

If S matches an element of S_2 there is a more complicated situation. If S matches a first element and n = 3 then a check is made to see whether T matches the second element of the same production. If this is true subroutine REPL4 is called to carry out operation (iv) and then return. If not then

REPL2 is called as above, and again if n = 3 the top element of S_2 must be popped for recondiseration and the parameter set to 2 (second element matched but not the first).

Subroutine REPL 4(n)

This subroutine carries out operation (iv) when the first and second elements of PRØDN match the first and second elements of a production in S_2. Since both productions of length one have the same right hand side, these are pushed into the stack S_u of critical productions. When there is a sequence of productions in S_u with the same right-hand sides then to signal the end of the sequence, the flag bit is set on the first byte of the final production of the sequence. Finally the zero-th element of the matched production in S_2 is replaced by the created symbol (which of course is still the topmost element in the symbol table $\mathcal{OT}$). Note that a push into a stack from PRØDN is assumed to be a non-destructive store from PRØDN.

Subroutine REPL2 (Y,X)

This subroutine combines the replacements required in operations (ii) (iii) and (iv). If the contents of PRØDN is a production of length two (i.e., T is not empty) then the subroutine REPL3 (Y, X) is called to make the appropriate replacement of the matched element in PRØDN, and set up the contents of PRØDN for the remaining replacements which is the same situation as if the contents of PRØDN was production of length one and a match had been found in S_2. The contents of PRØDN are pushed into S_u, a new symbol is created, the matched symbol in S_2 is replaced by the created symbol, the zero-th element of PRØDN is replaced by the created symbol (the first element is still the matched element since the store from PRØDN has been assumed non-destructive), and the contents of PRØDN is pushed into S_u (with flag bit set on first byte since this is the last production of this sequence of critical productions) before making a return.

Subroutine REPL3 (Y, X)

This subroutine replaces the matched symbol (parameter Y) with a created symbol, pushes the production in PRØDN into S_2, and the PRØDN is set up as a production of length one for the required replacements of productions of length one. The created symbol is placed in the zero-th position of PRØDN and the matched element (taken from the matched element in S_2 this time) is placed in the first position.

Subroutine SS1(Y)

This subroutine searches the stacks of productions of length one (namely S_1 and S_u) to find the element which matches the parameter Y. It in turn calls three further subroutines:

a) REPL3 - to handle the production of length two as above,

b) ADJS1 - to adjust the stack S_1 of the matching element is found here,

c) ADJSU - to adjust the stack S_u if the matching element is found here.

The stack S_1 is searched from the top down. If a match is found and PRØDN contains a production of length two (i.e., T is not empty), then subroutine REPL3 is called to push the modified production of length two into S_2 (according to operation (iii)), and then ADJS1 is called to modify the stack S_1 since an element has been matched in this stack. If PRØDN is of length one then only ADJS1 is called to adjust the stack S_1.

If no match is found in S_1, then the critical productions in S_u are searched for a match. When the match is found then, if T is not empty REPL3 is called to deal with the production of length two as before, and then ADJSU is called to adjust the stack S_u since an element in this stack has been matched. Note that if no match is found in this stack then all the stacks S_2, S_1 and S_u have been searched without finding a matching element. This is necessarily an error condition (ERRØR 2) since these subroutines are called only if the symbol of the production under consideration is a right-hand side which matches a symbol in the symbol table which is also a right hand side. Thus a match must occur in one of these three stacks.

Subroutine ADJS1(n)

This subroutine is called when the n-th element of S_1 is matched. Since this implies that there are two productions of length one with identical right-hand sides, both of these productions must be pushed into the stack S_u of critical productions. Thus the n-th production in S_1 is popped from S_1 and pushed into S_u. The productions of S_1 which were on top of the n-th and thus also popped in this process, are pushed back into S_1. The contents of PRØDN are pushed into S_u and the flag bit set since this is the end of this sequence of critical productions.

Subroutine ADJSU(n)

This subroutine is called when the n-th production of S_u is matched. Since the search of S_u was made from the top down, all the productions of S_u above the n-th are popped. Since the top element of the stack is now the final production of this sequence of critical productions, the flag bit signifying this on the top production of the stack is set to zero and is reset on the new top element of the stack after PRØDN has been pushed into S_u. Finally the productions of S_u which were previously popped are pushed back into S_u and a return is made.

3. DETERMINATION OF CONTEXTUAL CONDITIONS

In this second part of the translator generator system, the contextual conditions necessary to solve the parsing and ambiguity problems are obtained. Within given bounds, M and N, of right and left context respectively, we set up successively, context compatible with the local ambiguities determined from the simple production system obtained in the first part. Note that if the production system is not uniquely determined within the given bounds, it may be necessary to run the program again with one or both of the bounds increased. The notation of the first part is carried over into this section.

3.1 Basic Concepts

Since every CHOMSKY system can be embedded into a simple CHOMSKY system such that the set of words for Z in the simple system, over the alphabet of the original system, is the same as the set of words for Z in the original system (see [1]), it is sufficient to solve the problems raised in the introduction, for simple CHOMSKY systems only. Therefore, in the following we shall deal only with simple CHOMSKY systems as defined in definition 7. Note that this implies that

$$\mathcal{A} = \{S_i\} \cup \{T_i\} \cup \{U_j\} \cup \mathcal{A}^{\perp}$$

is a disjoint partition for $\mathcal{A}$.

Simple CHOMSKY systems have the valuable advantage, that all critical productions of Π belong to the set

$$\{B_j :: = U_j;\ j = 1,2, \ldots m\}$$

and that the local ambiguities are exactly the right sides of the critical productions which are contained in this set. This means especially that the local ambiguities of simple CHOMSKY systems are all of length one.

A necessary condition for a string $u \in \mathcal{L}(\mathcal{A})$ to be in γ is that u does not contain a substring XY of one of the following forms:

(a) $X \in \{S_i\}$, $Y \in \mathcal{A}^{\perp}$

(b) $X \in \mathcal{A}^{\perp}$, $Y \in \{T_i\}$

(c) $X \in \{S_i\}$, $Y \in \{T_i\}$ and $\nexists$ k

such that $XY = S_k T_k$

i.e., X,Y are not corresponding elements in the sets $\{S_i\}$, $\{T_i\}$ respectively.

For then it is apparent that u does not belong to γ.

In order to accomplish the determination of the contextual conditions we consider a five-tuple associated with each critical or undetermined production. Suppose C :: = V is a critical production. Then the five-tuple associated with this production is denoted by

$$(\alpha, V, \beta, W, C)$$

where α is the string of left context elements,

β is the string of right context elements,

and W is an element such that α V β is a word for W; i.e., there exists a sequence of productions in the production system Π, such that the string α V β may be reduced to W by making the indicated replacements.

An examination of W will determine what the next context element is since if $W = S_i$ for some i in { 1,2 ..., ℓ } then the next context element will be the corresponding T_i and thus T_i is added to the

right context string β. On the other hand if $W = T_i$ for some i in $\{1,2, \ldots, \ell\}$ then the next context element is the corresponding element S_i and S_i is added to the left context string α.

In order to determine whether a given critical production has been determined by the context found at a given stage we need to define the concept of compatibility.

Definition 8 (Compatibility)

Let $\alpha = A_{|\alpha|} \ldots A_2 A_1$

$\alpha' = A'_{|\alpha'|} \ldots A_2' A_1'$

$\beta = B_1 B_2 \ldots B_{|\beta|}$

$\beta' = B_1' B_2' \ldots B'_{|\beta'|}$ be strings.

then we call (α, V, β) compatible with (α', V', β') if and only if

(1) $V = V'$

(2) $A_i = A_i' \quad [\, i = 1,2, \ldots, \min(|\alpha|, |\alpha'|)]$

(3) $B_j = B_j' \quad [\, j = 1,2, \ldots, \min(|\beta|, |\beta'|)]$

i.e., the two strings α V β and α' V' β' are compatible if their common substring exhausts the smaller of the substrings of right context and also the smaller of the substrings of left context.

3.2 Implementation

3.2.1 Functional Description

3.2.1.1 Program Structure

The main program consists of three basic procedures called ZETA, MU and SIGMA which are utilized at each step until the set of undetermined productions vanishes. The bounds M, N of left and right context respectively are used as global parameters which stop the iteration if the set of undetermined productions has not vanished after (M + N) steps.

The procedure ZETA decides whether the context set up so far is sufficient for the determination of the critical productions. The procedure MU essentially makes all possible replacements in the productions by considering only those productions of length one. The procedure SIGMA obtains the context elements of the productions.

The procedure ZETA gives a partition of the set, G, of undetermined productions from the previous step into a determined part and an undetermined part. For a given production in this set, we determine whether there is different production (*) in the set which is compatible with the given production. If such a production (*) is found, then if W = W* the production (*) is placed in a subset G_A of G. If $W \neq W^*$ then the production (*) is placed in the set, G_U, of undetermined productions. The set of determined productions are those elements of G which are not in G_U.

The procedure SIGMA determines the context elements at a given stage, by checking whether the element W is a member of the set of the right-hand sides of the productions in the stack S_2 of productions of

length 2. If $W = T_i$ for some i, then we replace the string α by the string $S_i\alpha$, and replace W by a_i. If $W = S_j$ for some j, then we replace the string β by $\beta\ T_j$ and replace W by a_j.

The procedure MU operates on the element W of the five-tuple, and determines whether W is a member of the set of right-hand sides of the productions in S_2, or is a member of the set A_T of elements which do not appear on the right-hand side of any production. If not then an iteration process is entered. First all possible reductions are made from all the productions of length one, and then the resulting elements are again checked to determine whether they are right-hand sides in S_2 or in A_T. If not, we then repeat the process. A global parameter K is used to stop the process if the procedure is not satisfied after K steps.

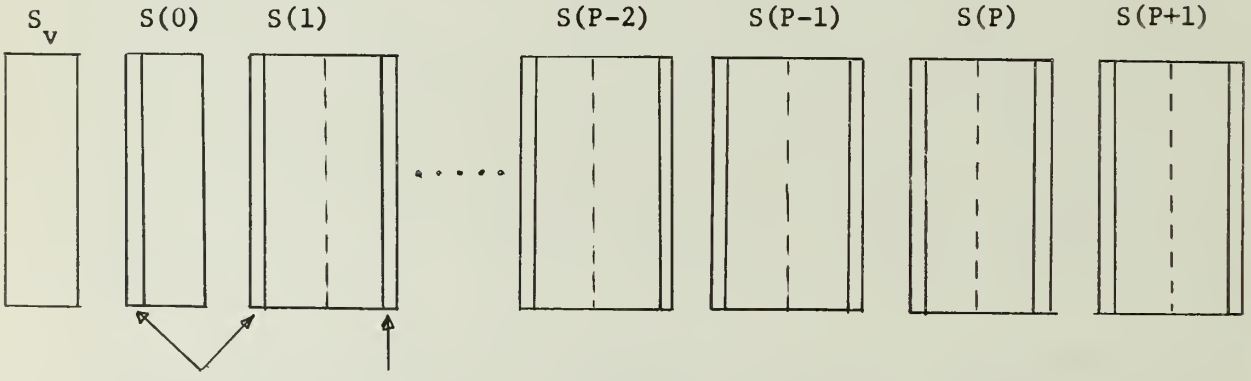
The object of the whole process is to reduce the set of undetermined productions to the empty set. However, if this is not achieved then one can either run the program again with an increased value for one or all of the global parameters M, N, K; or one can apply the production system as it is and accept the disadvantage of parsing strings in ways which do not lead to valid constructions in the language.

3.2.1.2 Data Structure

It is assumed that the global parameters M,N,K are given to the system, as are the results of the first part. The context is built up one element at a time (either left or right context) within the given bounds M,N and controlled by the variable P.

In order that the bounds may be considered finite but unbounded, the main idea of the implementation is that the context elements at each stage are stored in one stack which is linked to the elements of the preceding stacks in the sequence, in a cascading manner. The stacks in this sequence are denoted by S(0), S(1), S(2), ... S(P), S(P + 1). The pointer for the stack S(j) is denoted by p(j) and the k-th element of the stack S(j) is denoted by γ_k^j. Each element of every stack in the sequence after S(0) (i.e., for S(j), $1 \leq j \leq P + 1$) is to be at least two bytes in length, although extended where necessary to accomodate a larger internal number (see first part). The flag bit on the first byte of each element is called "FLAG 1" and the flag bit on the second byte is called "FLAG 2".

When the context parameter has the value P, we are finding the (P - 1)st contextual element, and we have P + 2 stacks.



are the typical elements in each stack

The first two stacks, S_v and S(0) have elements which are only one byte in length but extended when necessary as described in the first part. The first stack contains all the different elements V_k which are the right-hand sides of the critical productions found in the first part. The second stack S(0) contains all the left-hand sides of the critical productions. How these stacks are linked will be described below.

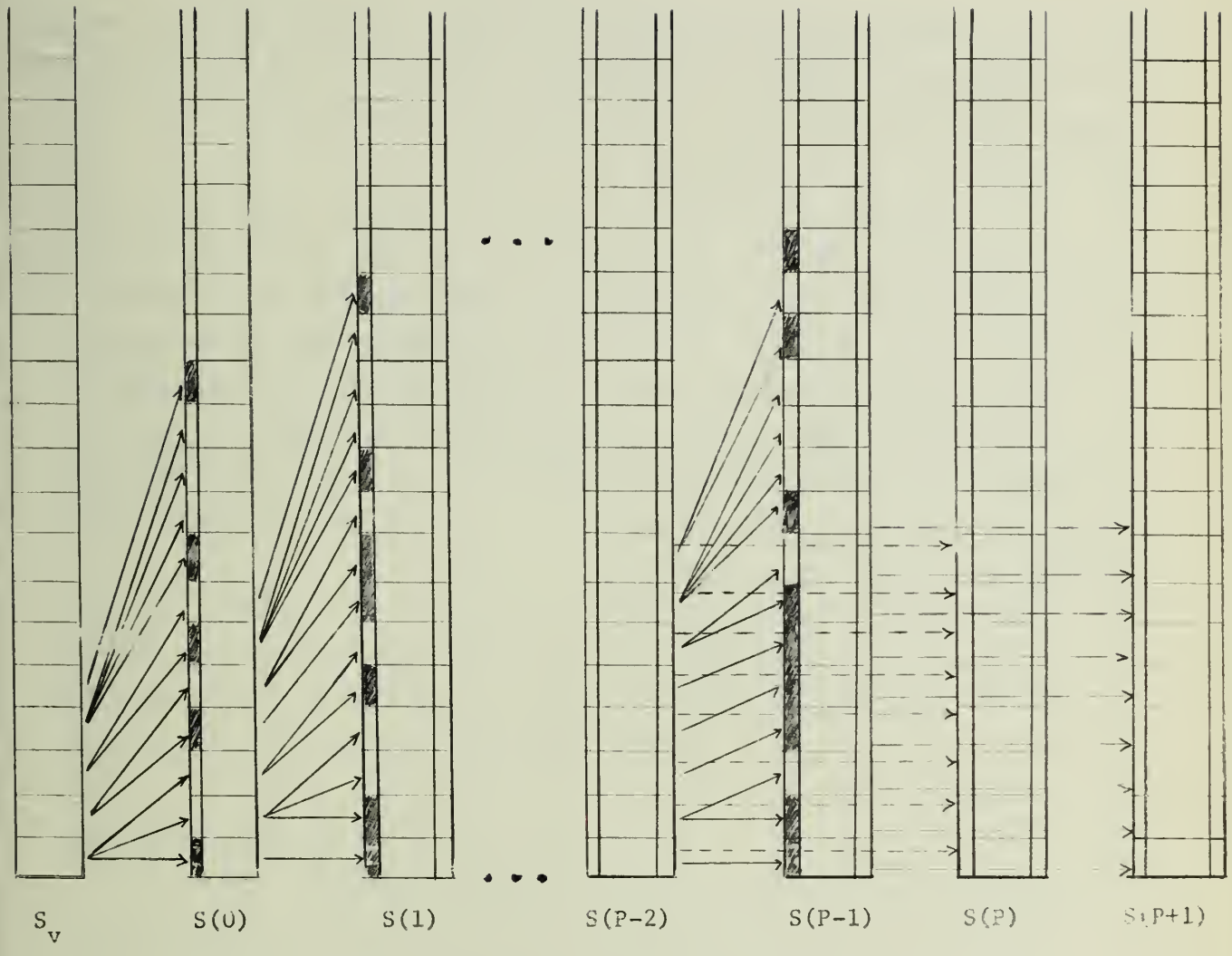
The first context element found for each production is stored in the stack S(1). If it is a left context element (i.e., $\gamma^\circ = A_1$) then FLAG 2 is not set, but if it is a right context element (i.e., $\gamma^\circ = B_1$) then FLAG 2 is set.

When a given production becomes determined, it is popped from this system and the context for the remaining undetermined productions is built up by building a new stack at each stage. When the context parameter has the value P we have the following situation:

> The last context element found for each production is stored in the stack S(P - 2). During this iterative pass the new context elements found are stored in S(P - 1). With each production is associated a five-tuple containing an element W, and these W's found during the previous iterative pass are stored in S(P) while the elements W being determined during this iterative pass are stored successively in S(P + 1).

Now in the stack S_u of critical productions found in the first part, for each right-hand side V_k, there exists at least one other production in S_u with the same right-hand side -- by definition of a critical production. Thus conversely, associated with each distinct V_k in S_u, there is a set of C_j's which contains at least two elements. Thus we first push an element V_k into the stack S_v and then push the set of C_j's associated with it, into the stack S(0), setting FLAG 1 on the first element of this set pushed into S(0). Thus to determine all the C_j's associated with the first element of S_v we must search the stack S(0) from

the bottom until we come to the second flag -- at which point we have a left-hand side of V_2, the second element of the stack S_v. In calling the procedure MU each such C_j could in itself give rise to such a set of left-hand sides, so we have 'cascaded' the set of productions corresponding to the right-hand sides V_k. Each of these can have its own first context element and so we push into S(1) the set of first context elements (the first element of each set having FLAG 1 set on) corresponding to each C_j. This cascading process is continued throughout the sequence of stacks of context elements. However, there is a one-to-one correspondence between the stack of elements W and the last stack of context elements at each stage. This means that there exists a one-one correspondence between the stacks S(P) and S(P-2) and also between the stacks S(P + 1) and S(P - 1). Thus the flags 1 and 2 have no significance in the stacks S(P) and S(P + 1) and hence FLAG 1 in these stacks has been utilized to denote that the flagged production is an element of the subset G_A of the set G of undetermined productions at the previous stage as mentioned in the previous section of this report.


S(0)
S(1)
S(P-2)
S(P-1)
S(P)
S(P+1)
FLAG 2 implies context element γ belongs to the right context string β
FLAG 1 implies production is in G_A
FLAG 1 implies the first element of a new group

Note that unless specifically denoted otherwise a PUSH or assignment operation is assumed to apply only to the bits of the byte itself and not to the flag bits as well. A pop operation of course will clear the entire byte (or bytes) including the flag bits.

The elements of the left context, α, and the right context, β, are stored in stacks S_α and S_β respectively. These stacks are built only for the production under consideration. However in order to determine compatibility we need two of each of these stacks -- one for the given production and one for the production being compared. We denote these respectively by $S_{\alpha 1}$, $S_{\alpha 2}$ and by $S_{\beta 1}$, $S_{\beta 2}$. The k-th element of $S_{\alpha 1}$ is denoted by α_{1k} and its pointer is denoted by $p_{\alpha 1}$. The k-th element of $S_{\alpha 2}$ is denoted by α_{2k} and its pointer by $p_{\alpha 2}$. Similar notation holds for $S_{\beta 1}$ and $S_{\beta 2}$.

3.2.2 Operation Description

The Main Procedure CONDETN (M,N,K)

The system is initialized by setting up the stacks S_v and S(0) as described above. In addition the left-hand sides of the critical productions (i.e., the C_J's) are also pushed into the stack S(2) as these are the first W elements of the five-tuple associated with each production. The counter GA is initialized to zero. This counter counts the number of elements in the ambiguous class G_A. Then setting the context parameter P to 1 we recursively call the procedure ZETA, which, in this implementation, itself calls the MU and SIGMA routines mentioned above. After calling ZETA, if the stack of undetermined productions, S_v, is not empty then if the P - 1 context elements found by this stage is less than the upper bound of M + N context elements, the parameter P is increased by one and ZETA is called again.

The initialization also includes the extraction of the anti-terminal symbols from the symbol table $\mathcal{A}$ in order to facilitate checking for the property of not appearing on the right-hand side of a production. These elements are stored in a stack A_T whose i-th element is designated by L_i and whose pointer is designated by P_ℓ.

Subroutine ZETA (P)

The running variable for the stack S(j) is denoted by i(j). Since the stacks are searched from the bottom up, the variables i(j) for j = 0, ..., P are initialized to zero. In order to facilitate returning to the next production in the sequence after comparison operations later in the program, the current values of each i(j) is saved in the location q(j). Then a call is made to the routine LP which builds up the context strings α, β for each production in the stacks $S_{\alpha 1}$ and $S_{\beta 1}$ respectively. In order to determine compatibility, for each production in the main pass through the set of undetermined productions, we must compare all other productions in this set. The 'switch' f is used to denote these different passes -- f = 1 denotes the main pass and f = 2 denotes the pass required to find a compatible production.

During consideration of the possible context strings for a given production, we may find that a given production becomes "determined" in the sense described above. In this case it is necessary to POP it from the system of stacks and this condition is noted by setting the 'switch' D to 1. Thus, on return from the consideration of all possible context strings for a given production in the procedure LP, if D = 1 the current element of the stack S(0) must be popped and the FLAG 1 denoting the first element of a new group adjusted according. The next production is now considered (if the last one was popped, the running variable is not increased by 1 since the next production is now in the position of the last) and the process is repeated. Upon exhausting the stack S(0) we return to the main program and consider whether it is necessary to repeat the process and find another context element. Note that if FLAG 1 is set on the next element of S(0) then we must now consider the next element of S_v and so its running variable, k, is increased by 1. If this stack becomes exhausted (by implication before S(0) is exhausted), then an error must have occurred in the organization of the FLAG 1's in S(0) and the ERROR call is made.

Subroutine LP (e,f,P)

This subroutine builds and keeps track of the stacks $S_{\alpha f}$, $S_{\beta f}$ containing the elements of the strings α, β respectively for each production. It has parameters e, the running variable; f, the value of which determines whether the production under consideration is one in the main pass or in the comparison pass; and of course the parameter P determining how many context elements have been found at this stage.

During the first iteration the parameter P has the value 2. Since all productions are undetermined by definition during the first iteration, we omit the procedure ZETA as described in section 3.3.1.1, and call the subroutine MU directly. Since the variables m, n denote the number of left and right context-elements respectively for each production, these are set to zero during this first iteration.

During other iterations we proceed through the stacks S(j) successively, using the current value e(j) of the running variable for each stack, determining whether the value stored in each stack is a member of α or β and so building up the stacks $S_{\alpha f}$ and $S_{\beta f}$ appropriately. If the last context stack S(P-2) is exhausted, control is returned to the calling point of this subroutine, while if any other stack is exhausted we have an error situation in the FLAG 1's as described in the previous subroutine.

When all the context elements have been found for this production at this stage (i.e., we have reached S(P-2), then a call is made to COMP 1 if this is the main pass, or to COMP 2 if this is the comparison pass. COMP 1 initiates the comparison pass and COMP 2 determines whether there is a compatible production. On return from these subroutines, the stacks $S_{\alpha f}$, $S_{\beta f}$ are adjusted to be ready to accept next context elements of the production. Note that because of the cascading nature of the stacks described above we may have to pop a sequence of elements from these stacks if the FLAG 1's denoting a new group are set. If this cascade reaches right down to S(0) then we have to return and adjust the first two stacks in the subroutine ZETA.

If the production is "determined" (i.e., D = 1), then it must be popped as in the subroutine ZETA. However since the possible context strings for a production may have many elements in common (in the preceding stacks S(j)), we pop only those stacks containing elements which are unique i.e., in cascading back to S(0) we pop no further stacks after finding one in which the next element is not a member of a new group. The preceding elements of each stack must be replaced after each pop. After completing the pops, we return to the process where we left off after setting the "switch" D back to zero. This final pop is affected by the short subroutine PR.

Subroutine COMP 1

This subroutine initiates the comparison pass for a given production by setting the "switch" f to 2. The comparison pass is designed to find a production compatible to the given one. This sub-

routine achieves essentially those decisions described in procedure ZETA in section 3.2.1.1. Since the definition of compatibility requires the right-hand side V be the same, we need check only those productions 'cascading' from one element V of the stack S_V. Thus we initialize the running variables g(h) for h = 0, ..., P-2 to the values they had on beginning new groups when we began consideration of the new V_k. We compare all possible productions up to the next FLAG 1 in S(0) when we would be beginning a new group corresponding to the next V.

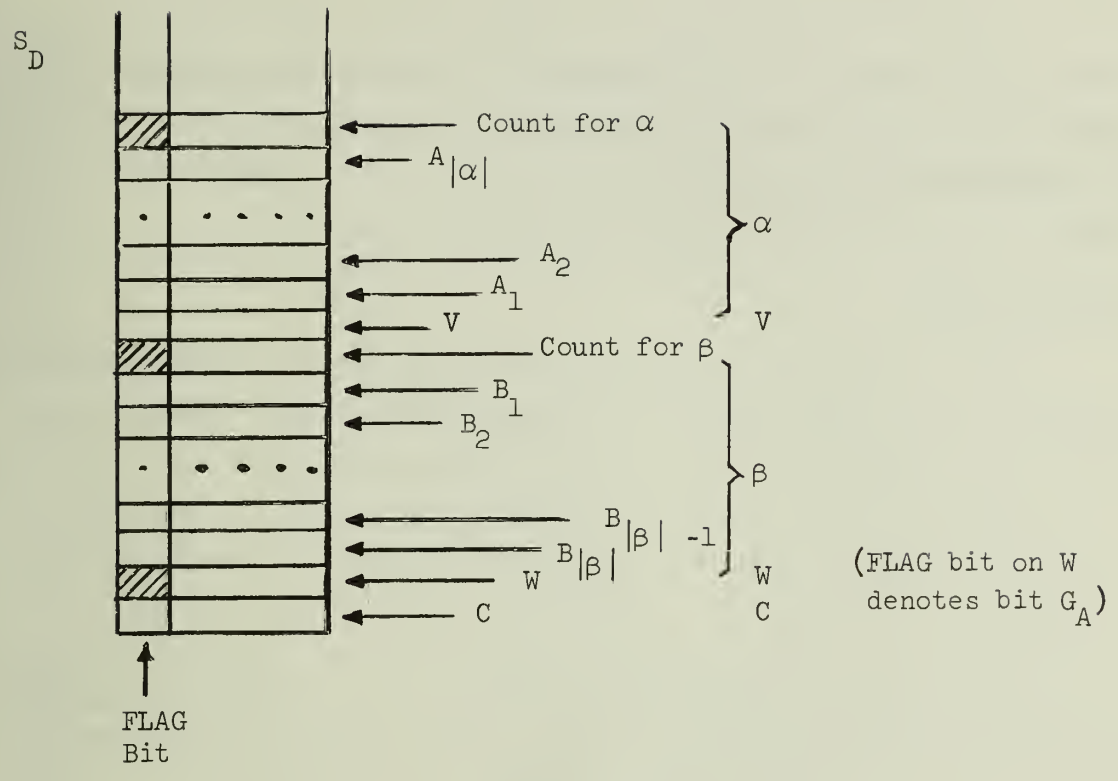
The number of elements in the α, β stacks corresponding to the given production are saved in variables m and u, n and v respectively for use later, and then we again make a call on the subroutine LP to build up the α and β strings for the productions being compared.

We noted in section 3.2.1.1 that the production is determined if it is not undertermined and so after checking all possible compatible productions and it is still not undetermined, it must be determined and we call the subroutine DET to place it in the stack S_D of determined productions. f is then set back to 1 and we proceed with consideration of the next production in the main pass.

Subroutine DET

If the production is found to be "determined", this subroutine pushes it into a stack S_D of determined productions and then sets D to 1 so that the offending production is then popped from the system.

In order to conserve storage, the 5-tuple corresponding to the determined production is stored "vertically" in the stack S_D whose elements are one byte in length, extended where necessary. The method of storage in S_D is depicted on the figure.



All productions in S_D are pushed in this fixed format and so unstacking will immediately produce the productions in the desired form viz:

$$W = \underbrace{A_{|\alpha|} \cdots A_2 A_1}_{\alpha}\ C\ \underbrace{B_1 B_2 \cdots B_{|\beta|}}_{\beta} :: = A_{|\alpha|} \cdots A_2 A_1\ V\ B_1 B_2 \cdots B_{|\beta|}$$

We first push in C from S(0) and then W from S(P). We check whether this production is a member of G_A (denoted by flag 1 set on the element W in S(P)) and if so call the subroutine DETA before popping the W element of this production from the stack S(P). The subroutine DETA determines whether this production is a valid member of G_A and if so increases the counter GA of the number of members in the ambiguous class G_A.

Next we push into S_X, in order, the elements of right context, the last found first and the first found last, from the stack S_β, followed by the count for β i.e., the number of elements of right context. The count is flagged, firstly to distinguish it from the internal numbers representing

symbols and secondly to denote the end of the arbitrary string β. V is pushed in next followed by the elements of the left context α, the first found element first and the last found element last, from the stack $S_{\alpha 1}$ followed by the count for α flagged as above.

Subroutine COMP 2

This subroutine determines whether this production in the comparison pass is compatible with the given production in the main pass. Since we have required $V = V^*$, recall that this means that

$$A_i = A_i^* \quad [i = 1, 2, \ldots, \min (|\alpha|, |\alpha^*|)]$$

$$B_j = B_j^* \quad [j = 1, 2, \ldots, \min (|\beta|, |\beta^*|)]$$

Since we know these conditions were satisfied for the previous iteration, we need only check that the last found context element satisfies these conditions and so we compare the top elements of the stacks S_α, S_β from each pass. If it is not compatible, we return and consider the next production of the comparison pass, while if it is compatible, we check that the left-hand sides, C, (found in S(O)) are different and then if the W elements (in S(P)) are the same we flag the W element of the given production to denote a member of G_A and return to the comparison pass, else if the W elements are different we have found an undetermined production and so make a call to the subroutine MU before initializing the α and β stacks of the comparison pass and returning to the next production of the main pass.

Subroutine MU

The subroutine MU places the W of the production being considered in a dummy location Q and then calls the main subroutine, OP, on the undetermined productions. The subroutine OP includes the procedure SIGMA described in the first section. The subroutine OP operates on the W of the production and determines whether it is 'open' i.e., whether it is a member of the set of elements which are on the right-hand side of the

productions in S_2 or it is in A_T (i.e., is not on the right-hand side of any production). If W is not open, it is returned with the flag bit set which means a call must be made to NUITER which is the subroutine which makes all possible replacements from the productions of length 1. Before returning from MU we set FLAG 1 on the next available cell in S(P-1) since the next context element placed in there will be the first element of a new group as we will then be considering the next W element in S(P).

Subroutine OP(x)

This subroutine determines whether the argument, X, is open (i.e., an element of a right-hand side of some production in S_2) or does not appear on any right-hand side. This latter case was the reason for putting all these anti-terminal elements in a separate stack A_T. If X is not open, we set the flag bit and return. However if X is a member of a right-hand side of S_2, we take this opportunity to determine the next context element. The number of left and right context elements found to date for this production have been saved by means of the variables m, n respectively and so after checking that the addition of another context element will not violate the given bound, the context element is pushed into the new context stack S(P-1) and FLAG 2 is set if it is a right context element. The left-hand side of the matched element in S_2 becomes the new W and so is pushed into the new W stack, S(P + 1). The next available cell of S(P-1) is then cleared of everything including flags so that a FLAG 1 set to denote a W in G_A will not cause confusion here. Note that by construction in part 1, all the right-hand sides of S_2 are different from each other and all other right-hand sides and so having found one match we need look no further.

If the addition of a further context element would violate the given bounds M or N of left and right context, then we can find no more context within the given bounds. In this case the production is pushed into a stack S_C of the same form as S_D described previously, and a decision about what to do about such productions must be made upon completion of the program. Clearly this problem could be eliminated by running the program again with increased bounds.

On the other hand if a match is obtained with an element of A_T, then no further reductions can be made since this element does not appear as the right-hand side of any production in our system. In this case we first check that this element is not the distinguished element Z of our language, and if it is not we push this production into a stack S_R, of the same form as S_D. If the element is the distinguished element, Z, then after checking that the addition of another context element would not violate the bounds on right context, we add the symbol # as the next right context element. The symbol # is the "string terminator" symbol and in this context means that replacing it by a right element at the end of a string is not possible without leading outside of Y. The modifications to the stack when a new context element is added are carried out as described above and in this case the new element W in the stack S(P + 1) remains as Z.

Subroutine NUITER(Q)

This subroutine is an iterative procedure making all possible reductions from the productions of length one if the argument Q is not open. The iteration is bounded by the given global parameter K, and if all the new W elements so produced are not open after K iterations, they are pushed into the new W stack S(P + 1) with the corresponding new context elements left blank and the process continues as before.

The iteration is controlled by the variable ℓ. We initially check Q against the right-hand sides of all the non-critical productions. If no match is found, it is compared to the right-hand sides of the critical productions. If no match is found here either, then we must have an error situation since Q was not open and by construction of part one

$$\mathcal{A} = \{S_i\} \cup \{T_i\} \cup \{U_j\} \cup \{V_k\} \cup \{A_T\}$$

is a disjoint partition of the alphabet $\mathcal{A}$.

If a match is found with U_j for some j then Q is replaced by the left-hand side of the matched production, a call is made to the subroutine OP with this new Q as argument and the iteration proceeds.

If a match is found with V_j for some j, the same procedure is followed except that now all possible left-hand sides associated with the V_j are now used as replacements successively. Note that if a given V_j is not matched then the running variable g of the stack S(O) of left-hand sides must be adjusted appropriately according to the scheme outlined previously. Note also that this running variable g must also be a function of the iteration variable ℓ, and hence having found an open Q, we must check that all the possible "left-hand sides" have been dealt with before returning from this subroutine. This is the reason for the series of tests before the return on page 9B of the flow-charts.

Subroutine DETA(X)

This subroutine determines whether or not the given production is a valid member of the ambiguous set G_A. This requires that the element W associated with this production is a word for the distinguished element Z. All those productions for which W is not a word for Z may be dropped from the set G_A.

The element W of this production is stored in the i (P)th position of the stack S(P) and it is placed in the cell W in order to carry out these operations. If W is the element Z, then the flag 1 associated with the W element in the stack S_X is set in order to denote that this production is a member of the class G_A and the counter GA (which counts the number of elements in the class G_A) is increased by 1 before returning to DET(X) to store the remaining elements of this production in S_X. But if W is not the element Z, we attempt to reduce it to Z by finding a sequence of replacements from the stack S_1 of productions of length 1, since W itself is just of length 1. If no further reductions can be found (the top of the stack S_1 is reached without finding a matching element for W), then this production may be dropped from the set G_A and we return to DET(X) as before.

4. PURPOSIVE PARSER

4.1 Implementation

4.1.1 Functional Description

The production system constructed in the first two parts allows the purposive parsing of strings; for this one uses a modification of the NU procedure used before. The production system consists of those productions stored in the stacks S_1 and S_2, which contain productions of length one and two respectively and also those in the stack S_D of 'determined' productions in which the sets of critical productions have been 'determined' or differentiated by context.

Let Z be the distinguished element which is given and suppose the program which is to be scanned is read into the stack S_N with elements X_i and with pointer p_N. The program string is terminated by reading the string terminator symbol, #, into the top of the stack. Assume the beginning of the program is at the bottom of the stack, and we assume a left to right scan. Since we do not assume a 'look back' facility and in order to simplify the procedure, we made only one replacement of a string element at each consideration of an element, it is necessary to iterate the scanning passes through the program string, the iteration being bounded by an arbitrary given parameter Q.

4.1.2 Program Description

The Scanning Procedure

We first initialize the iteration parameter q and the variable i which counts the elements in the program string. Note that if the program string x is 'open' (i.e., if $X_1 \in \{T_k\}$ or $X_{|x|} \in \{S_k\}$) then $x \notin \gamma$, i.e., the program is not a sentence in the language. Thus we first check whether $X_1 \in \{T_k\}$ and if so go to EXIT 1 which is an error condition since such an element needs an element of $\{S_k\}$ in front of it in order to be reduced.

We next note that if x = u U v and U is an anti-terminal element (i.e., $\in \mathcal{A}^1$ consisting of elements which do not appear on the right-hand side of any production) then $x \notin \gamma$, since no further reductions could then be made on this element and it could not thus be reduced to the distinguished element Z. This is of course assuming that Z is not the only element in the program string, in which case we have finished. If, upon reaching the top of the stack S_N (excluding the string terminator symbol #), and the single element is not Z or there is more than one element left in the stack, then we increase the iteration counter q by one and, provided this counter is less than the iteration bound Q, we return to the start and repeat the process. If the iteration bound has been reached, then we exit by EXIT 3 and must re-evaluate the situation. It may be necessary to increase Q.

Having checked these initial conditions on the element X_i of the program string, we proceed to finding the appropriate replacement rules in the procedure NU. X_i is first checked against the right-hand sides of the distinct productions of length one found in S_1. If a match is found, the appropriate replacement is made in the stack S_N and we return to consideration of the next element in the program string since the alphabet has been divided into disjoint sets as described in the previous parts.

If no match is found in S_1, we next seek a match in S_D. The method of storing these productions in S_D was explained in part two, and an examination of this structure will show why it is necessary to examine the flag bits on each element of the stack in order to jump over the context strings α and β for each production. Since the first flag is set on the count for the right context β, if a match is found and the count of β is zero, we need only compare the left context α. The number of left context elements is again found by checking for the flag bit, but note that if the count backwards, k, on the program string reaches zero, then there are not enough left context elements in the program string and this replacement rule fails. Otherwise we check each context element against the corresponding element in the program string. If a non-matching element is found, we return to the search of S_D but otherwise

on reaching the end of the left context string (denoted by another flag), a valid context has been found. Thus, since for this value of j we have $V_j = |\alpha|$, and the count for β is obtained from the element in S_N with index $j - V_j - 2$. With this new value of the index j we finally obtain the replacement symbol for this production from the element of S_N with index $j - V_j - 2$ [compare the storage scheme for S_D in previous part].

If the symbol X_i was not matched by a right-hand side of a production in S_D, then we must jump over the left context of this production in S_D, before returning to look for the next right-hand side to compare.

If the count for β was not zero after a match for a right-hand side has been found, then we must check also the right context (see label F). If the count, i, on the program string plus the count for β, V_j, is not less than the stack pointer p_N, then there are not enough right context elements in the string for this replacement to be made, and we resume the search on S_D. Otherwise we compare each right context element against the corresponding symbol in the program string as was done above for left context. If this comparison is successful, the left context is compared as above.

If no match is found in S_D, then we search for a match amongst the right-hand sides of productions of length two. If the entire stack S_2 is searched without finding a match, then we have an error exit at EXIT 2 since we should have found a match in one of these stacks due to the partition of the alphabet into disjoint classes. If the element X_i matches a member of the class $\{T_k\}$, then we return to the beginning and consider the next element of the program string. This is because the previous element may have been replaced to become an element of the class $\{S_k\}$, and we are not assuming a 'look back' property. This replacement would be made on the next scanning pass through the program string. If, however, a match is found with a member of the class $\{S_k\}$, then after checking that X_i is not the last element of the program string, we check that the following element of the program string is the corresponding member of this production in S_2. Note if X_i is the last element of the string, then the string is open and so we go to error

exit EXIT 1, since we cannot obtain the corresponding T_k that this final S_k needs in order to made a replacement. Otherwise, if the corresponding T_k does not match the next element of the program string, then it must be a member of one of the other disjoint classes of the alphabet and so we return to the beginning and consider this next element of the program string. However if it does match, then the two elements are replaced by the left-hand side of this production in S_2 and then remaining elements of the program string in S_N are pushed down one space, in turn, before a return is made to the beginning to consider the next element in S_N.

As mentioned previously, when the top of the stack S_N is reached, if there is more than one element remaining in the stack or if the single element is not the distinguished element Z, then the scanning counter q is increased by one and another scanning pass is initiated provided the scanning counter is less than the iteration bound Q. If the single remaining element in S_N is the distinguished element Z, then we have achieved our goal of reducing the program by successive applications of the productions of the system.

5. CONCLUDING REMARKS

5.1 Consequences of the Parsing Algorithm

In order to determine to what extent we have solved the ambiguity problem and to what extent this parsing algorithm is really purposive, we have to discuss the following cases:

(a) $S_U = \emptyset$ and $G_A = \emptyset$

Recall from 3.2.1.1 that G_A was a subset of the set, G, of undetermined productions at a given stage. An element of G_A has the property that if (α, V, β, W, C) is the 5-tuple associated with this production, then $\alpha V \beta$ is ambiguous in the following sense:

$$\exists \; i, j \text{ such that}$$

$$C_i :: = V_i \quad , \quad C_j :: = V_j$$

are productions in the production system Π such that

$$V = V_i = V_j \quad , \quad i \neq j,$$

$$\alpha V \beta \; \rho_o \left\{ \begin{matrix} \alpha \; C_i \; \beta \\ \alpha \; C_j \; \beta \end{matrix} \right\} \underline{\rho} \; W$$

In this case, for any $u \in \gamma$ (i.e., u a word for Z), the parsing algorithm gives, successively, strings u_i $(i = 1, 2, \ldots, n)$ such that $u = u_1$, $u_n = Z$, $u_i \; \rho \; u_{i+1}$. While running, the main program does not consider tentatively other sequences v_j which do not lead to Z.

The ambiguity problem is solved since Σ is unique. This is because it does not contain ambiguities in the sense described above, which are necessary for the existence of ambiguities (see definition 6)

(b) $S_u = \emptyset$, $G_A \neq \emptyset$

In this case we have ambiguities in the sense described in case (a) above. Such W have to be checked whether they are words for Z, i.e., W ρ Z. All elements of G_A for which the corresponding W is not a word for Z can be dropped from G_A, as is done in the procedure DET in the second part of this system.

The production system is ambiguous if and only if G_A, thus reduced, is not empty.

This means that the parsing algorithm should sometimes consider more than one sequence $\{u_j\}$, all of which lead to Z and therefore are of interest. However at the present time, the algorithm has been set up to consider just one sequence, namely that obtained by using the first matched production found, and the notation is given that the production system is ambiguous. This decision will be reviewed in the light of practical experience with the system but there would seem to be some doubt as to the practical utility of a system which is ambiguous in this sense.

(c) $S_u \neq \emptyset$

In this case the ambiguity problem is not solved. It may possibly be solved by increasing any or all of the bounds, M, N, K (See 3.2.1.1)

For the parsing problem, the comments under (a) and (b) above hold except that we generate terms u_i which contain a right side of a production which is not in S_u, i.e., a production in either S_1, S_2 or S_D. All the substrings u_i which are words for Z and which do not contain a substring which is a right side of a production in S_1, S_2 or S_D but which contain a substring which is a right side of a production in S_u, must be of the following form:

(i) u contains a substring $v \in \mathcal{L}(\mathcal{A} \sim \mathcal{A}^{\perp})$ with $|x| \geq m$, where m is the length of the right side of the critical production associated with the locally ambiguous string x, contained in the substring v. (See definition 5) i.e., v contains a locally ambiguous substring x which implies from the definition that the right side of a corresponding critical production must be a substring of i.e., $m \leq |x|$.

(ii) u contains a substring v' such that there exists a sequence

$$\{v_i\} \quad (i = 1, 2, \ldots, t) \quad , \quad v_1 = v', \; v_t \neq Z$$

i.e., there exist sequences which lead outside of γ, the set of words for Z. By increasing K and/or M and/or N, both of these situations become rarer.

If one of these situations occurs during the parsing of a string one has two choices:

(i) One can run the main program again with one or more of the bounds M, N, K increased, and thereby improve the resulting production system.

(ii) One can tentatively apply all the corresponding productions of S_u and accept the disadvantage of following sequences which lead outside of γ i.e., do not lead to the desired distinguished element Z.

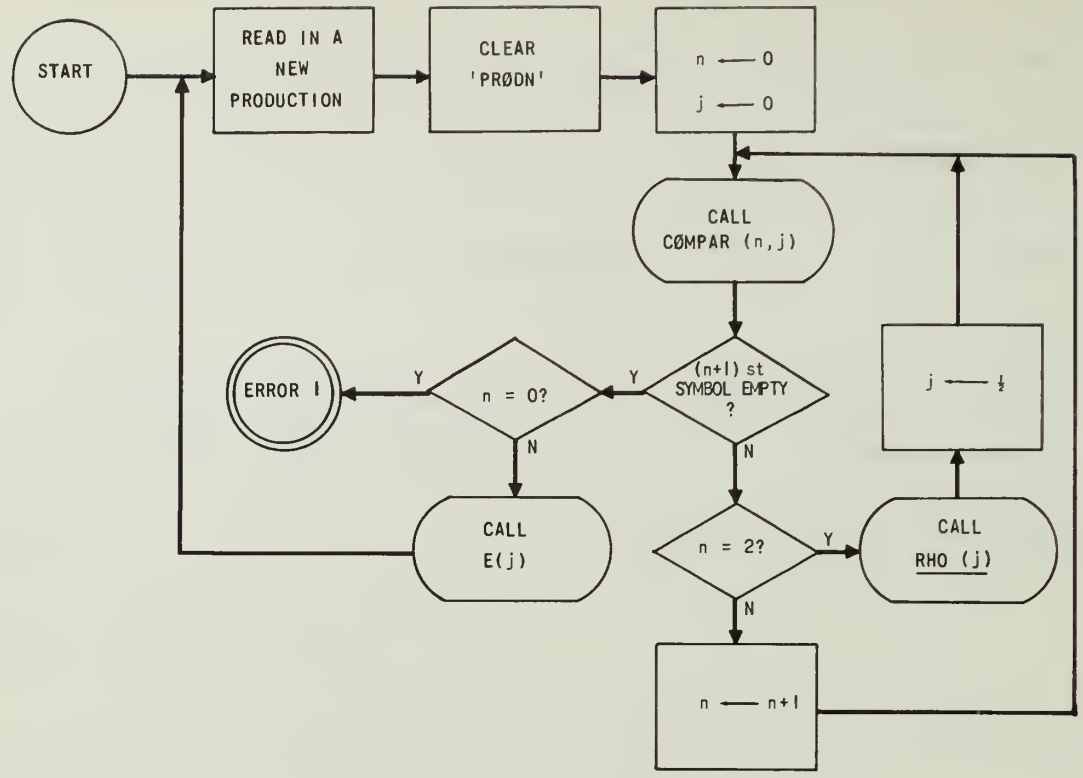
5.2 The Semantics

The next stage in the system is to adjoin the semantics to actually compile the code for the constructs recognized by the parsing algorithm. One method of doing this would be to attempt to match the semantics associated with the definition of the language in BNF, to the rather expanded set of productions produced by the main program. However it has been suggested [2] that an easier method would be to run each right side of the original definition of the language, through this system and hence obtain a definitive context for each of these. Then we use these original productions (with context) purposively in the scanner. In this form the semantics may be adjoined in their original form and we have also eliminated all the "created characters" used to "determine" the production system. However this will have to be the object of further study.

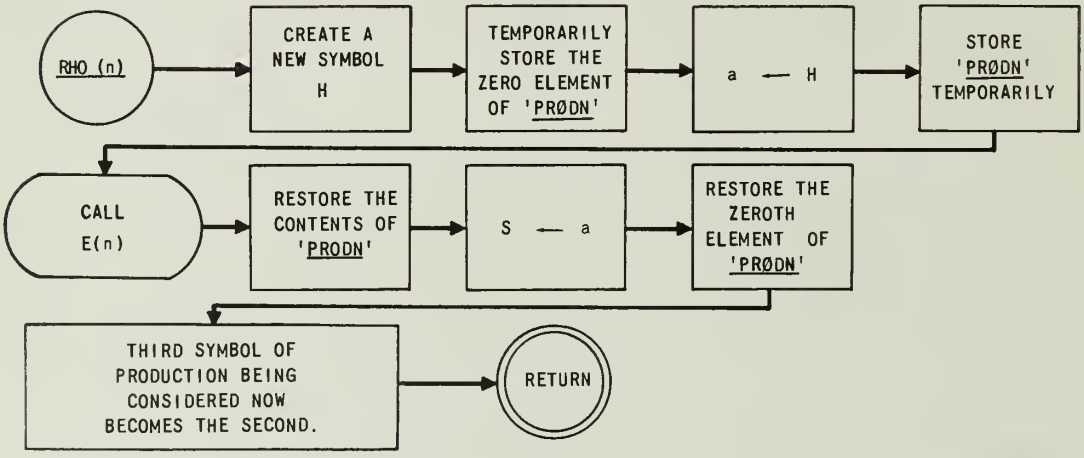
REFERENCES

[1] Eickel, J. and M. Paul, "The Parsing and Ambiguity Problem for Chomsky-Languages," TH Munchen, Bericht Nr. 6409, (August 1964)

[2] Paul, M., Private Communication

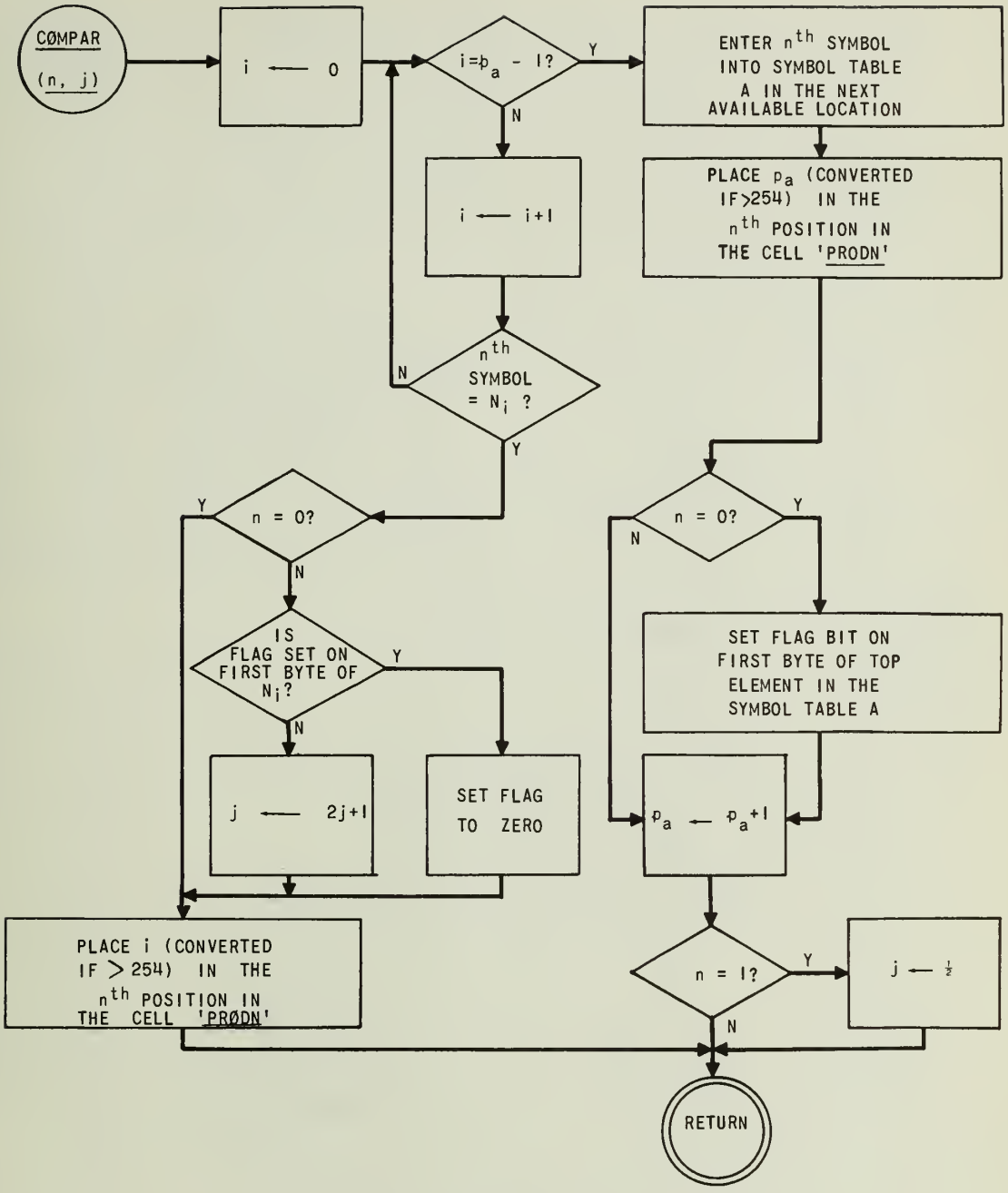


MAIN PROCEDURE: SIMPLE



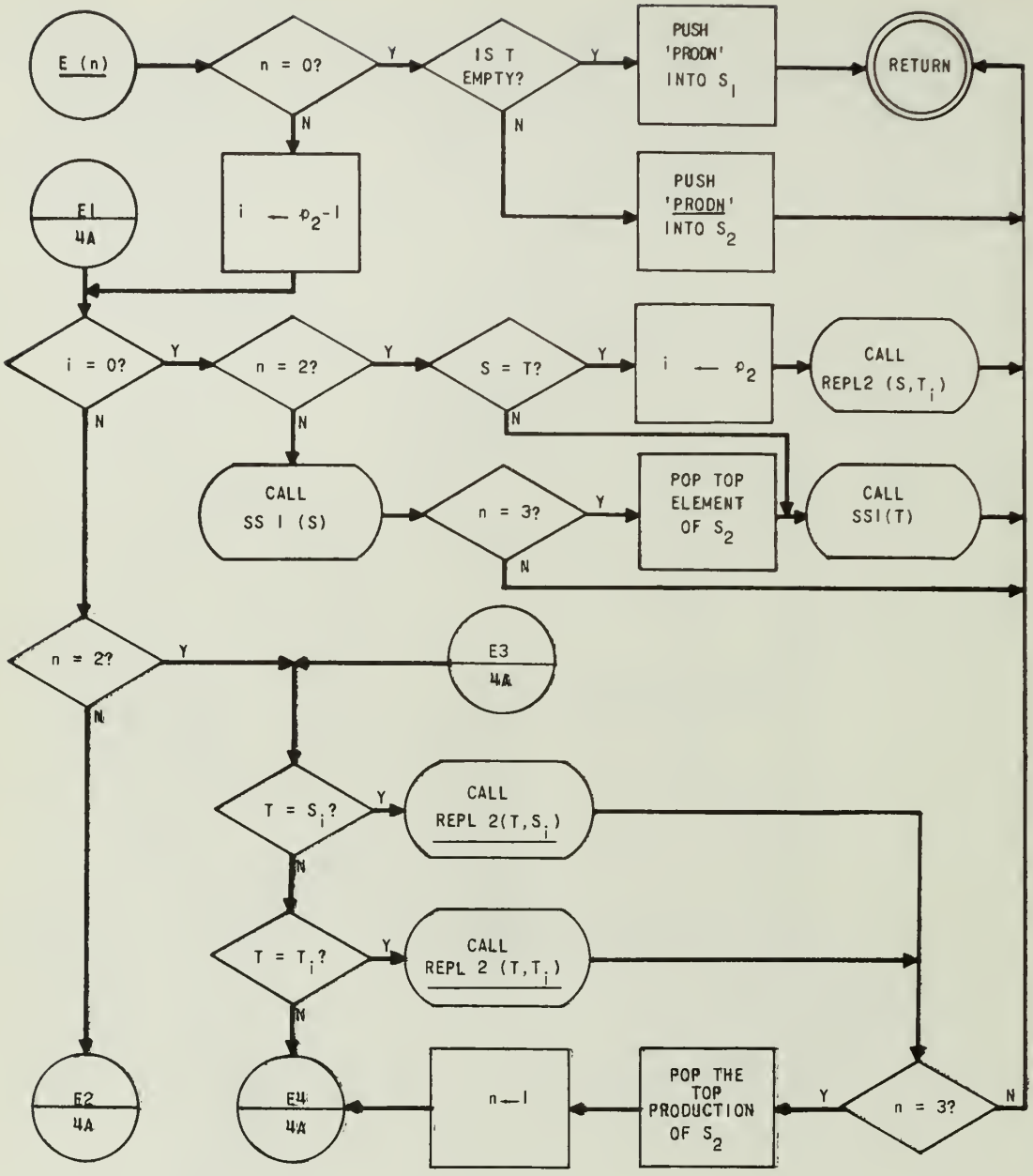
SUBROUTINE: RHO (n)

FIGURE 1A



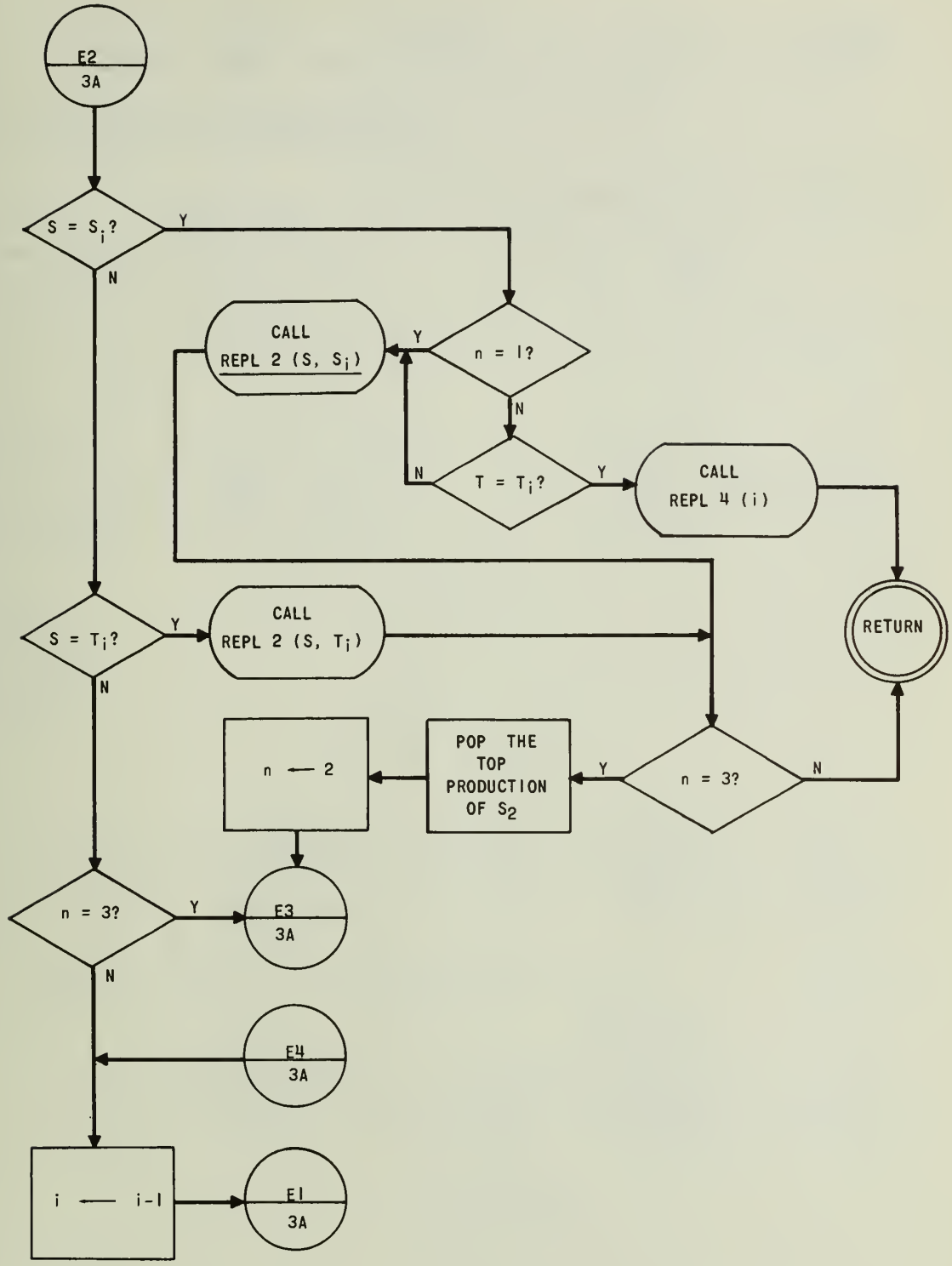
SUBROUTINE: CØMPAR (n,j)

FIGURE 2A



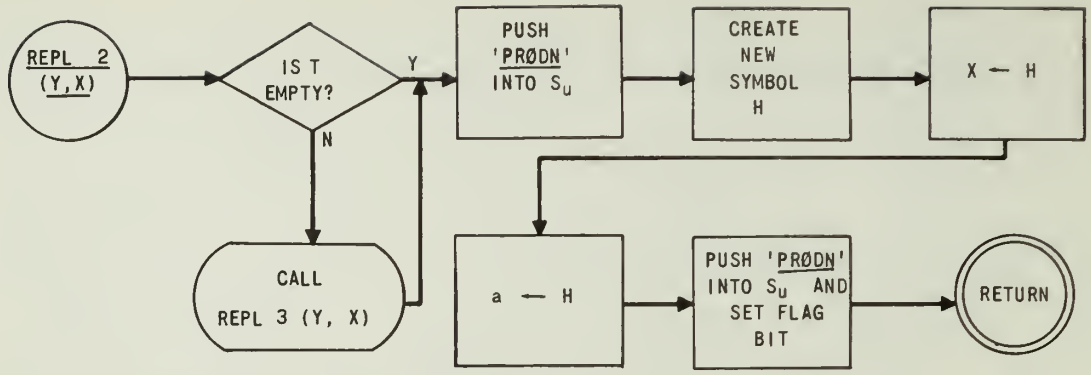
SUBROUTINE: E(n) [PAGE 1]

FIGURE 3A

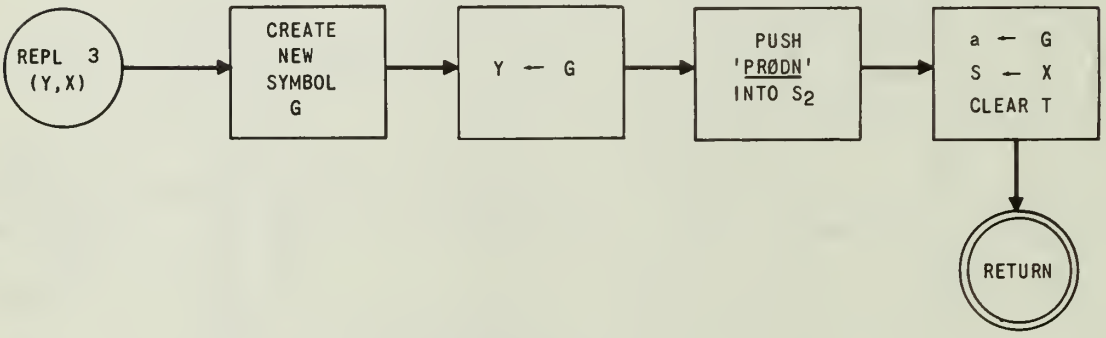


SUBROUTINE: E (n) [PAGE 2]

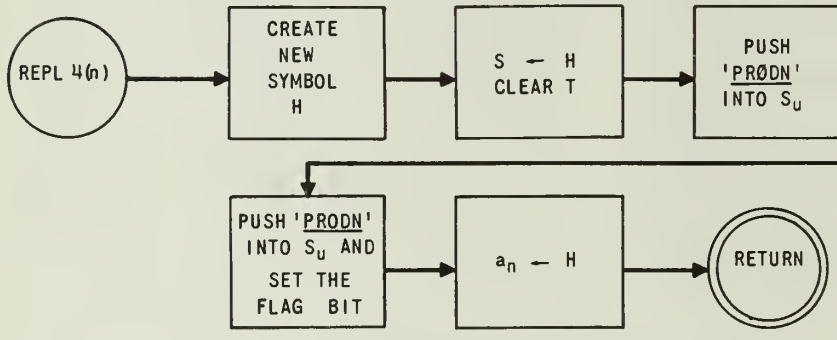
FIGURE 4A



SUBROUTINE: REPL 2 (Y,X)

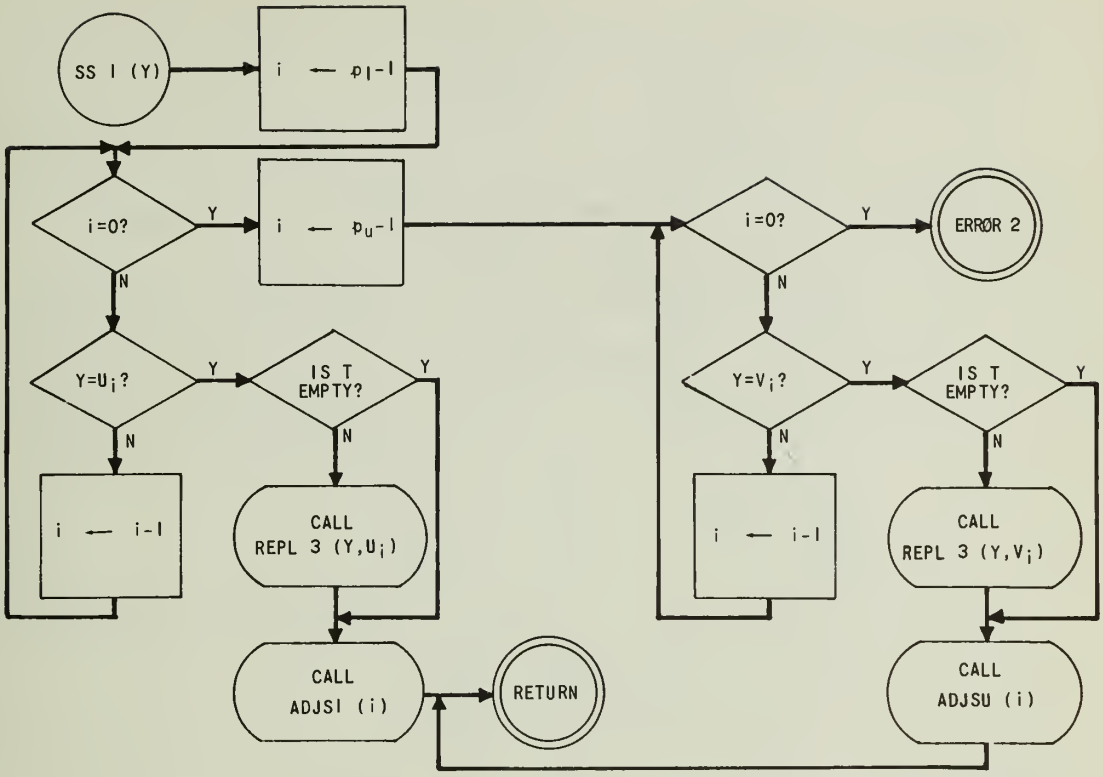


SUBROUTINE: REPL 3 (Y,X)

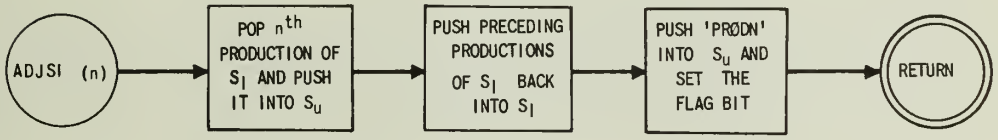


SUBROUTINE: REPL 4 (n)

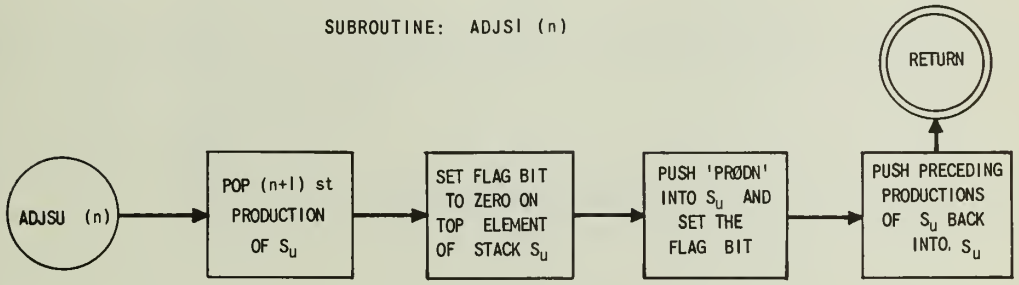
FIGURE 5A



SUBROUTINE: SS I (Y)

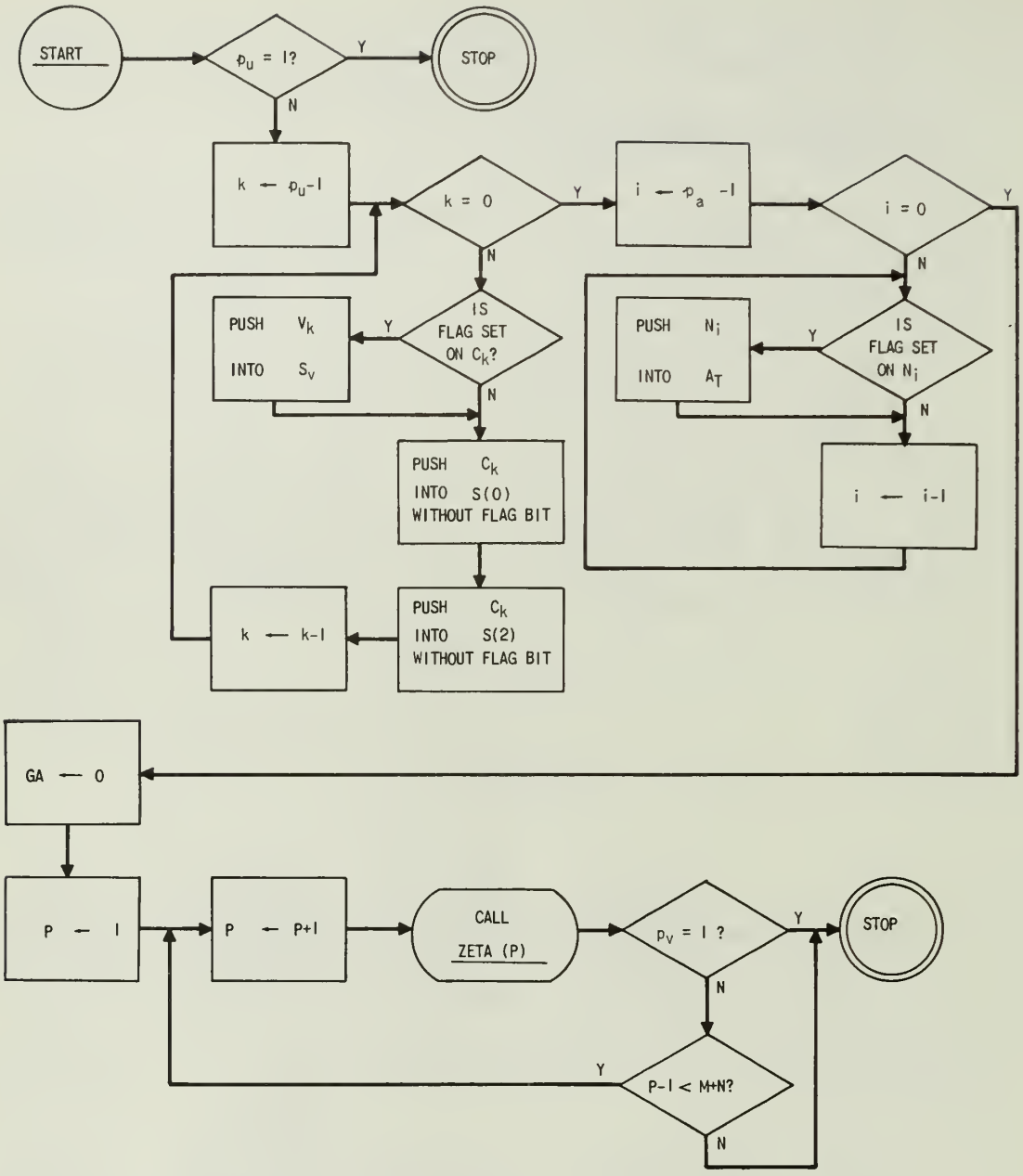


SUBROUTINE: ADJSI (n)



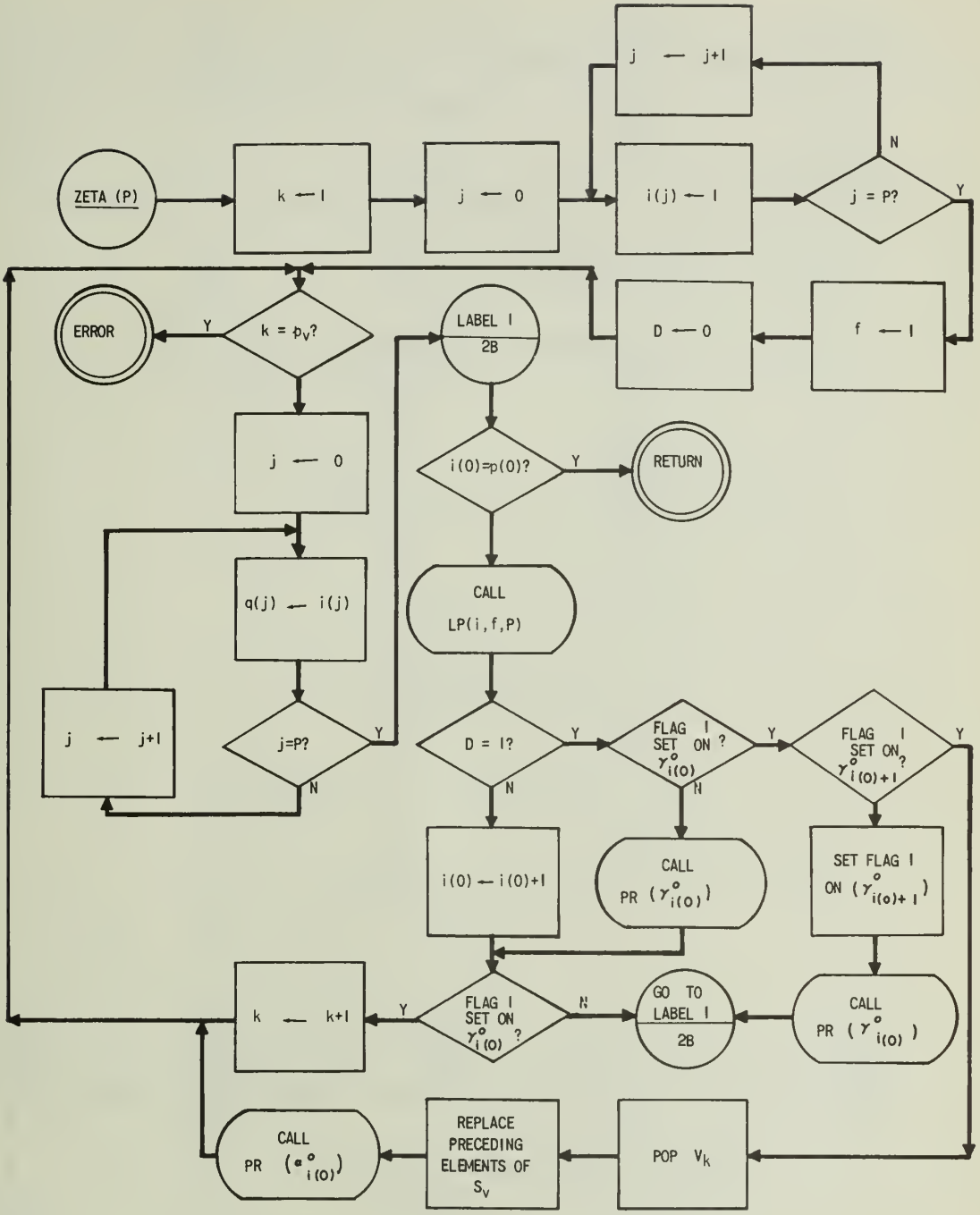
SUBROUTINE: ADJSU (n)

FIGURE 6A

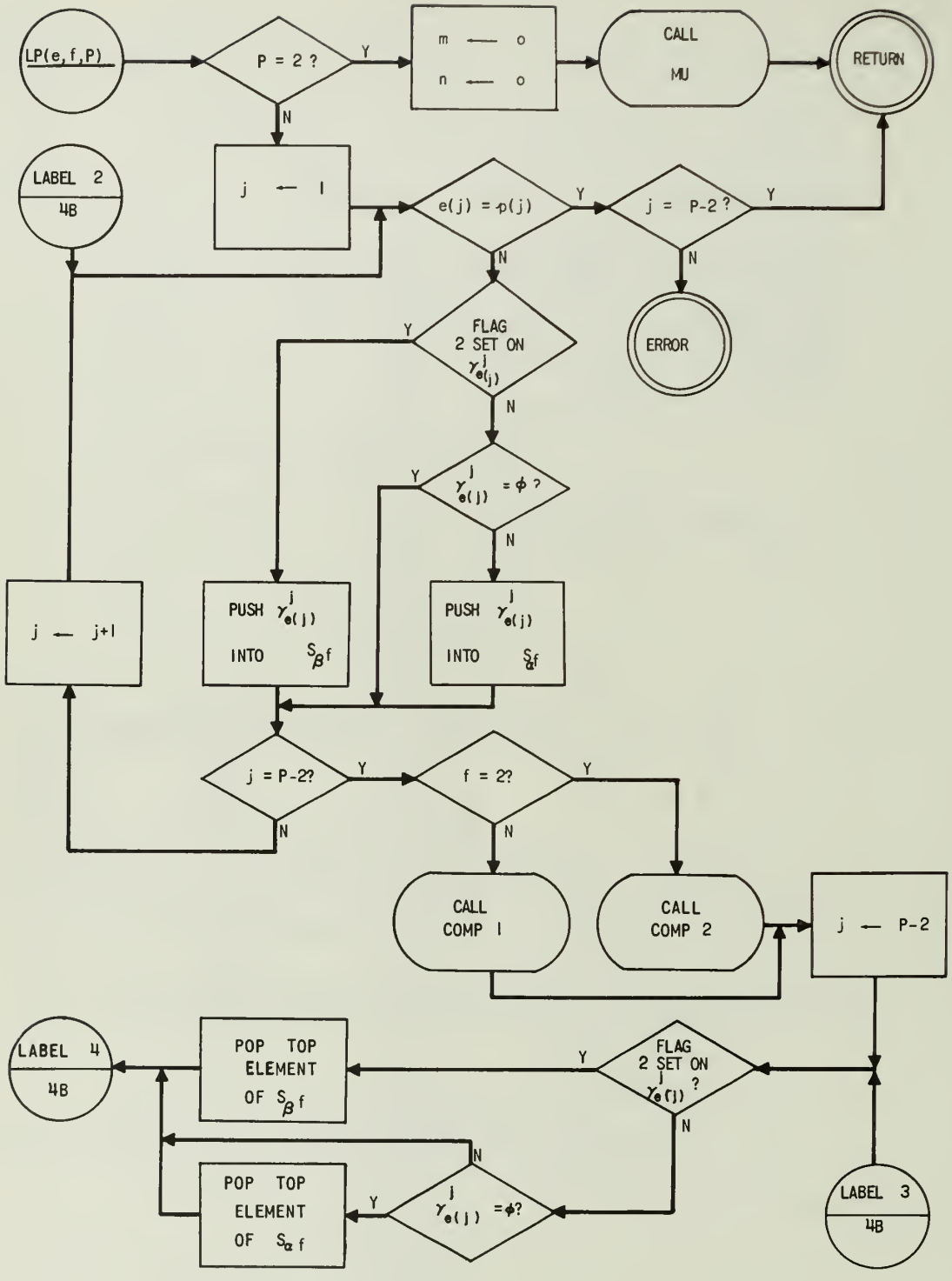


MAIN PROCEDURE: CONDETN (M,N,K)

FIGURE 1B

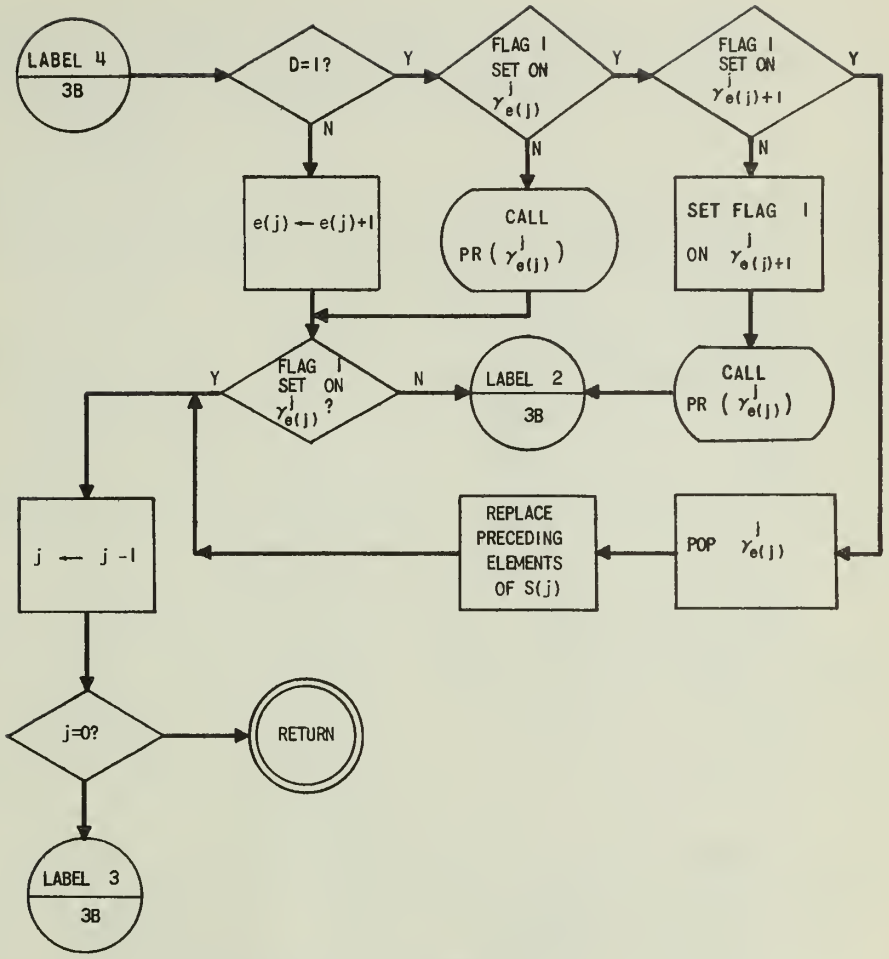


SUBROUTINE: ZETA (P)
FIGURE 2B

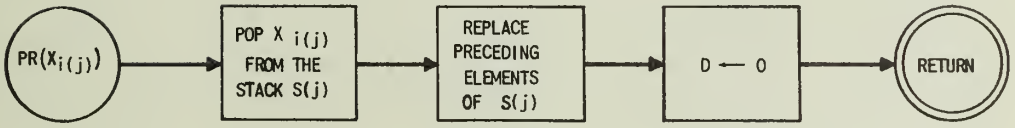


SUBROUTINE: LP (e,f,p) [PAGE 1]

FIGURE 3B

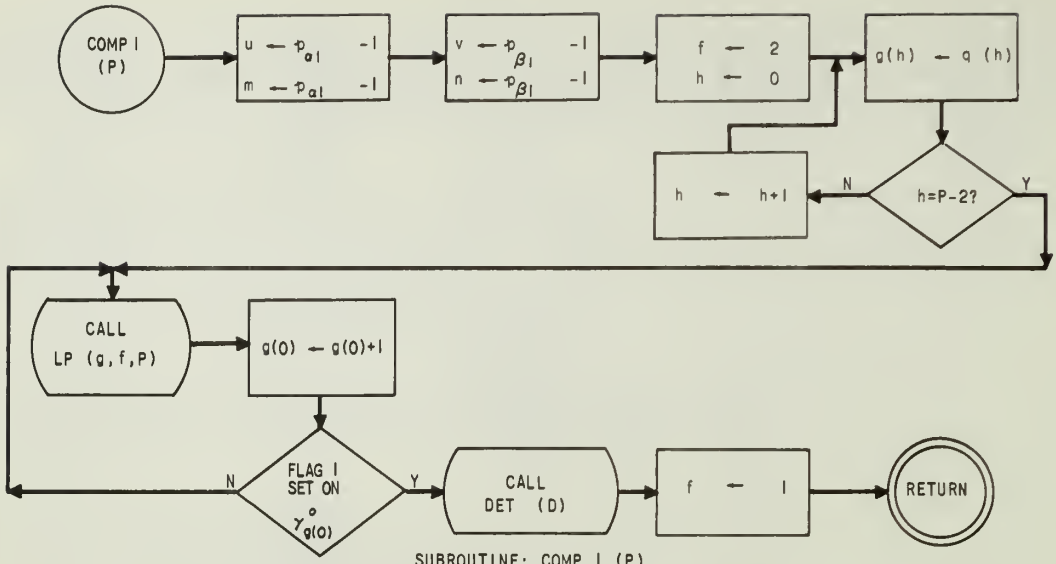


SUBROUTINE: LP (e,f,p) [PAGE 2]

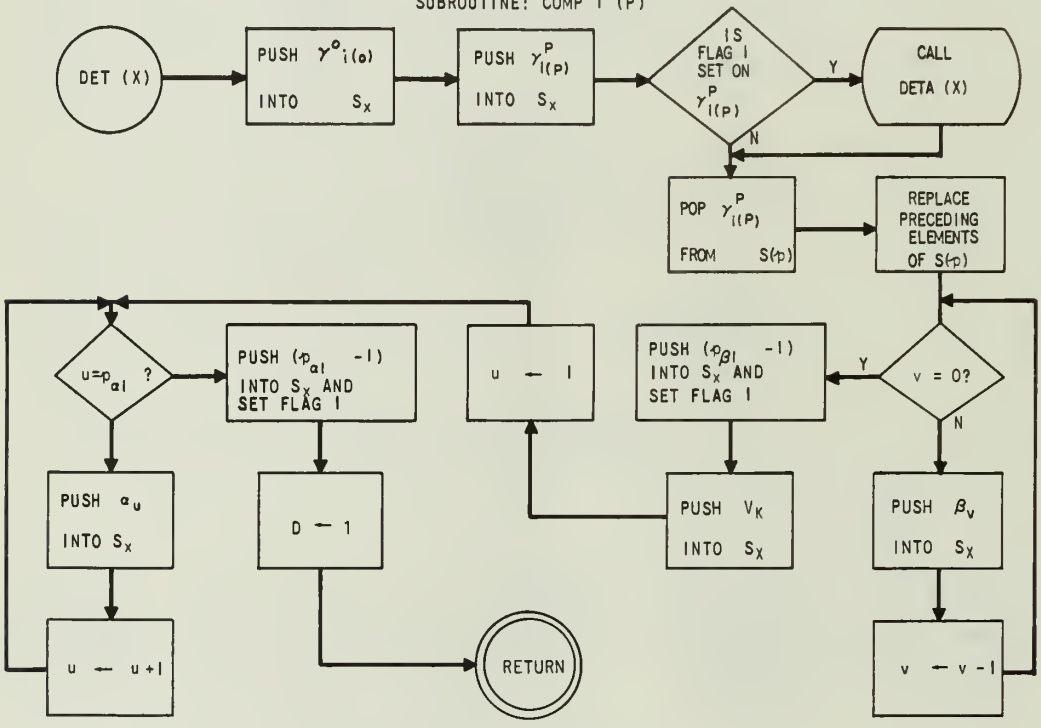


SUBROUTINE: PR ($X_{i(j)}$)

FIGURE 4B

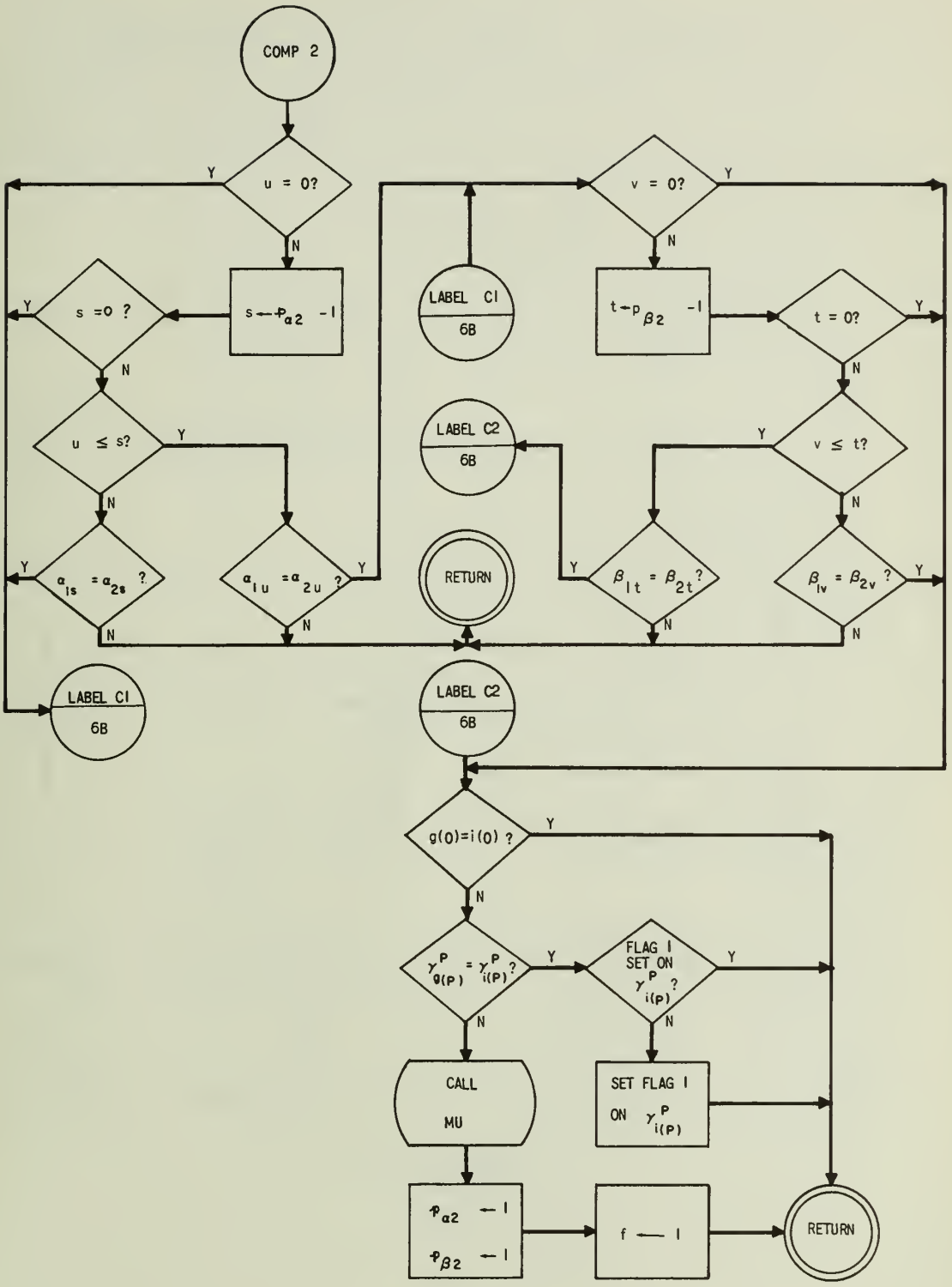


SUBROUTINE: COMP I (P)



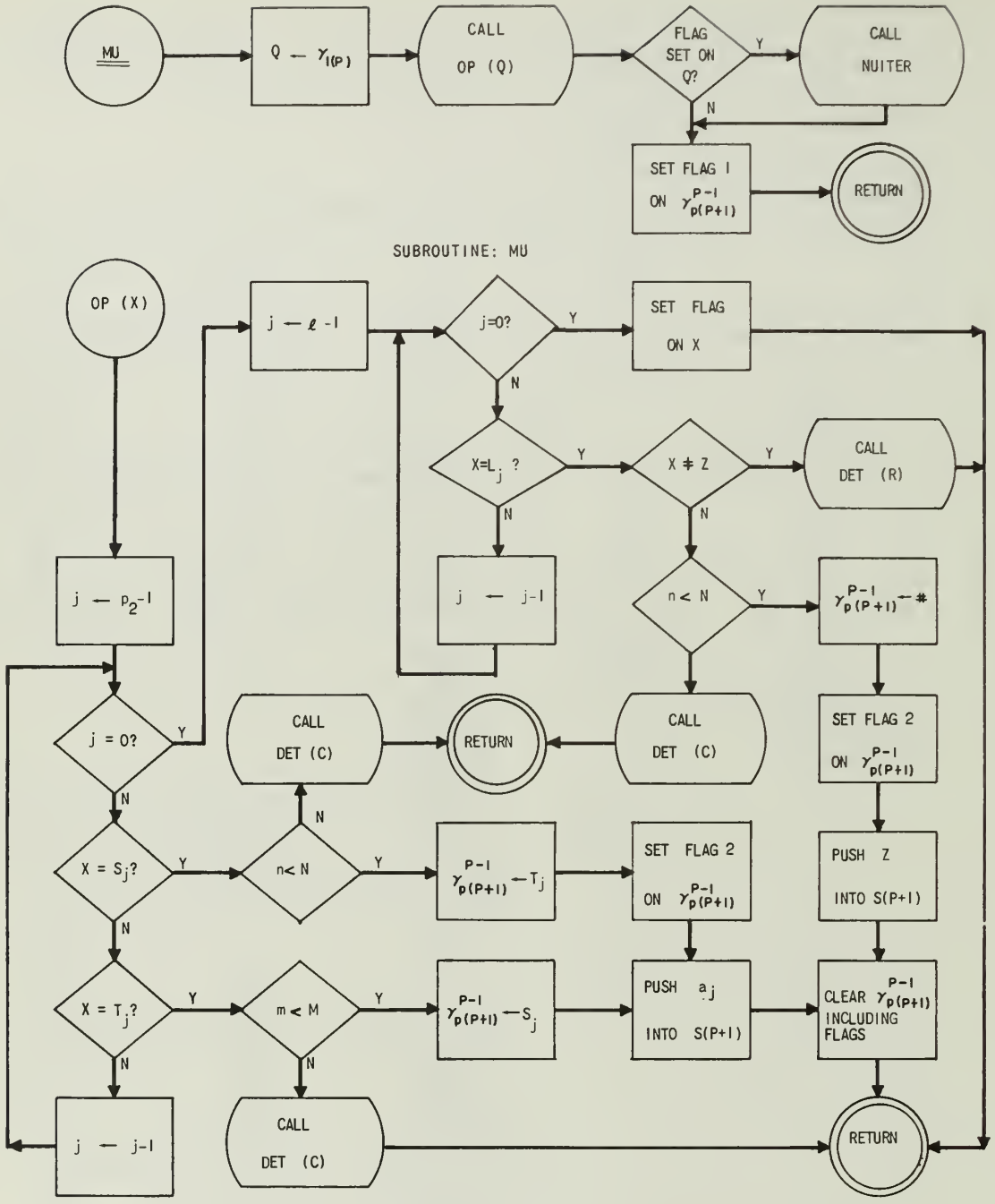
SUBROUTINE: DET (X)

FIGURE 5B



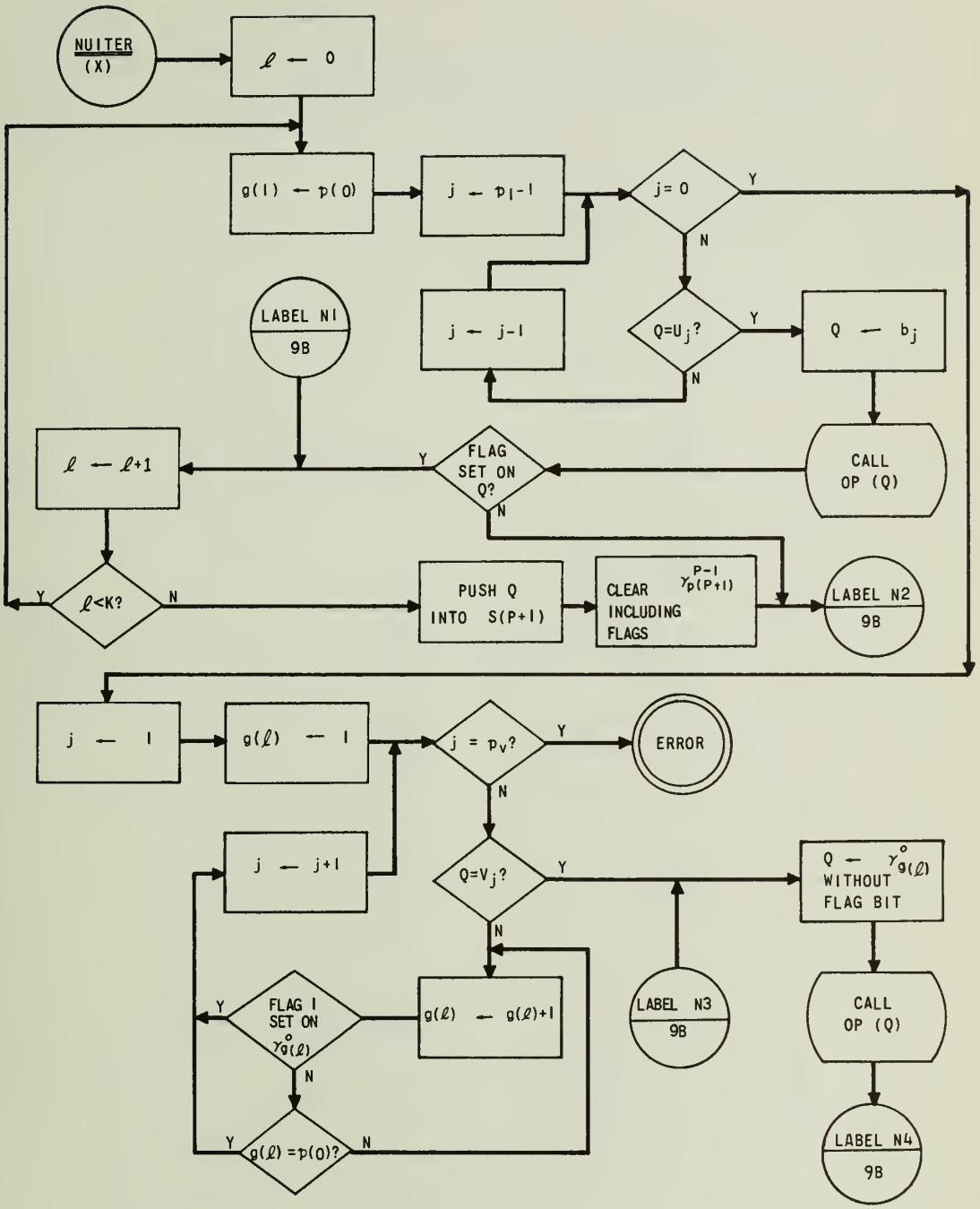
SUBROUTINE: COMP 2

FIGURE 6B

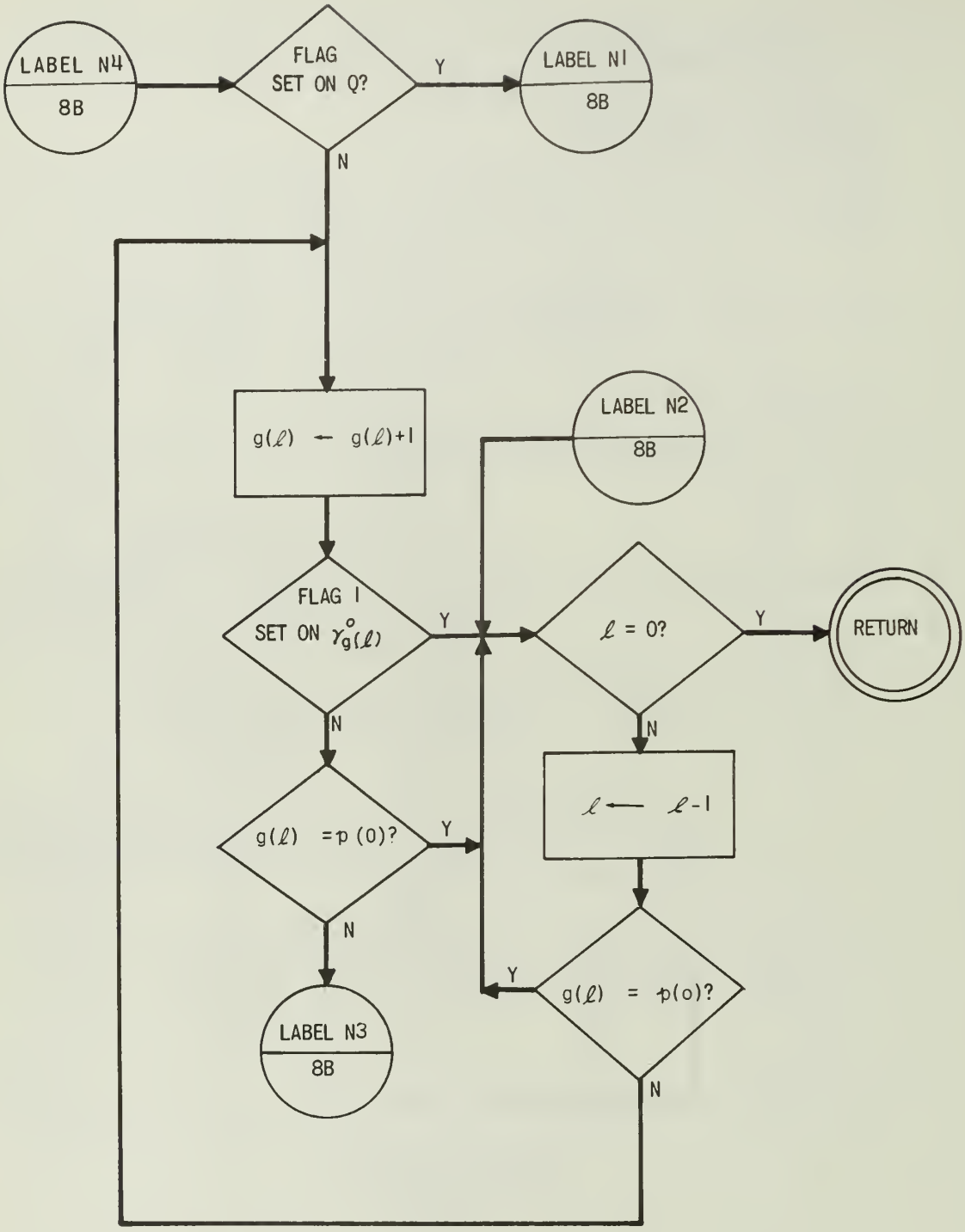


SUBROUTINE OP (X)

FIGURE 7B

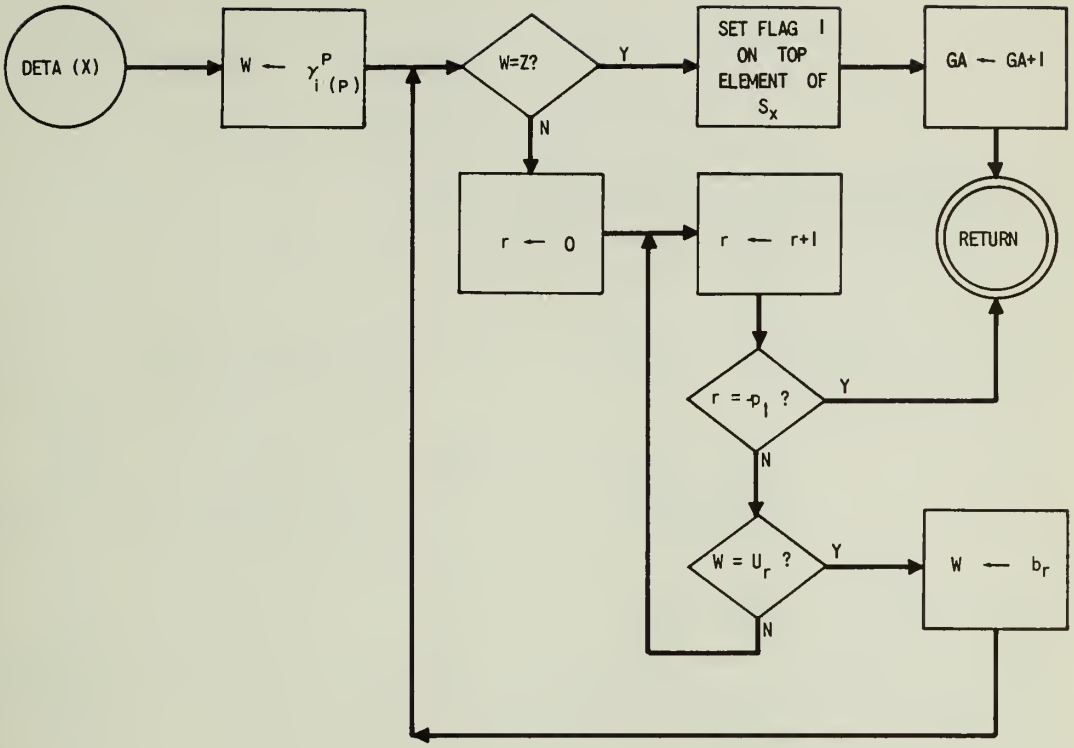


SUBROUTINE: NUITER [PAGE 1]
FIGURE 8B



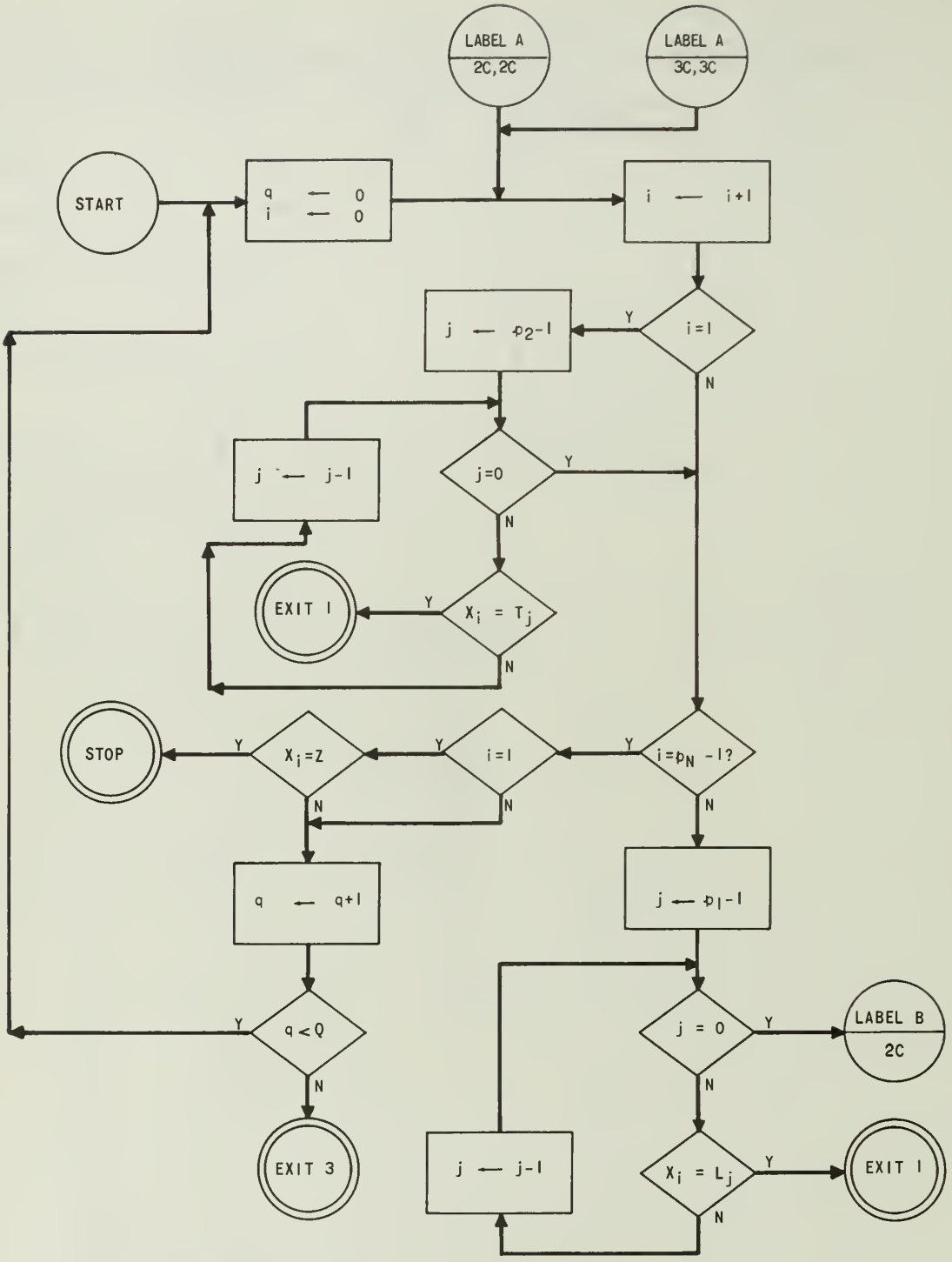
SUBROUTINE: NUITER [PAGE 2]

FIGURE 9B



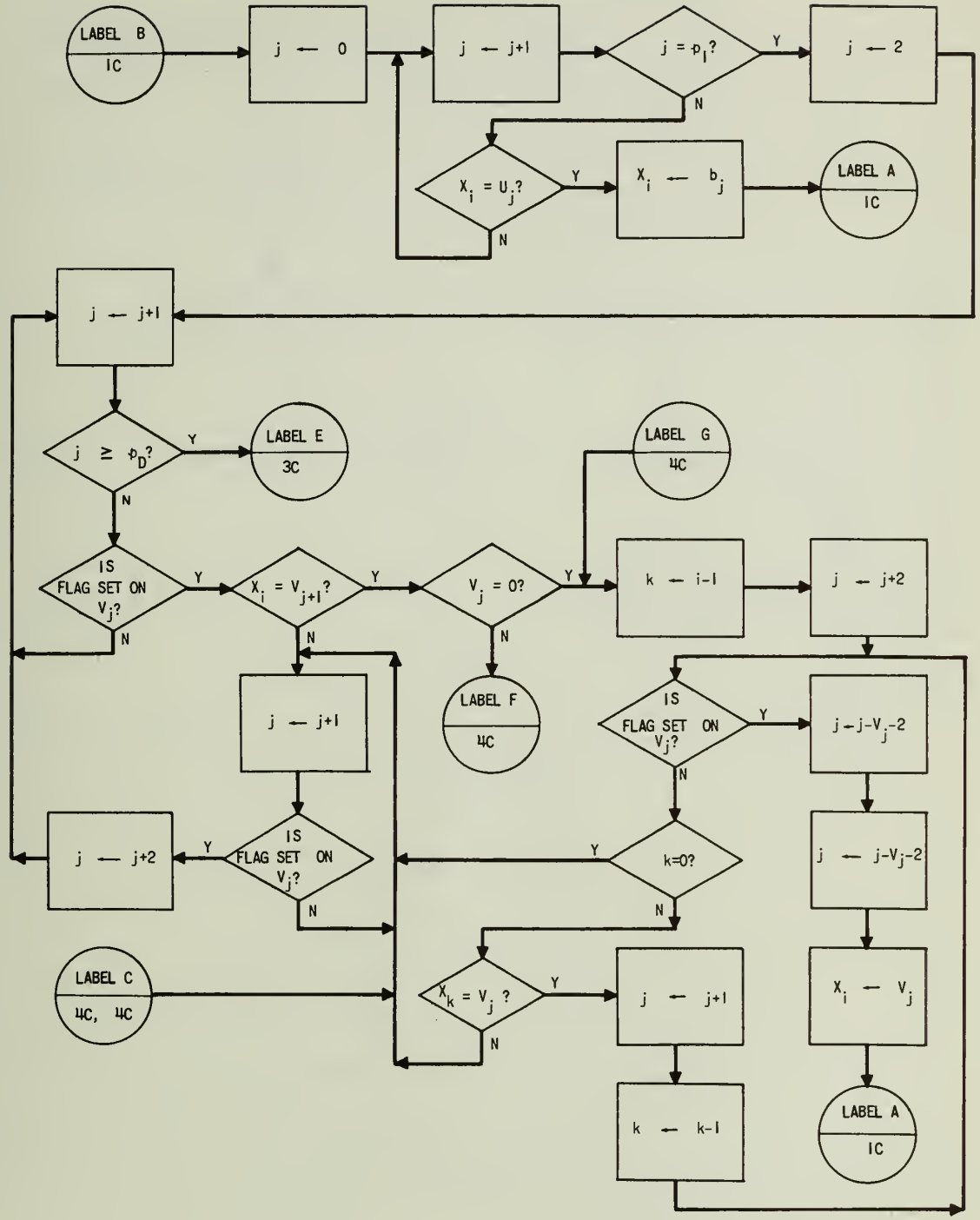
SUBROUTINE: DETA (X)

FIGURE 10B

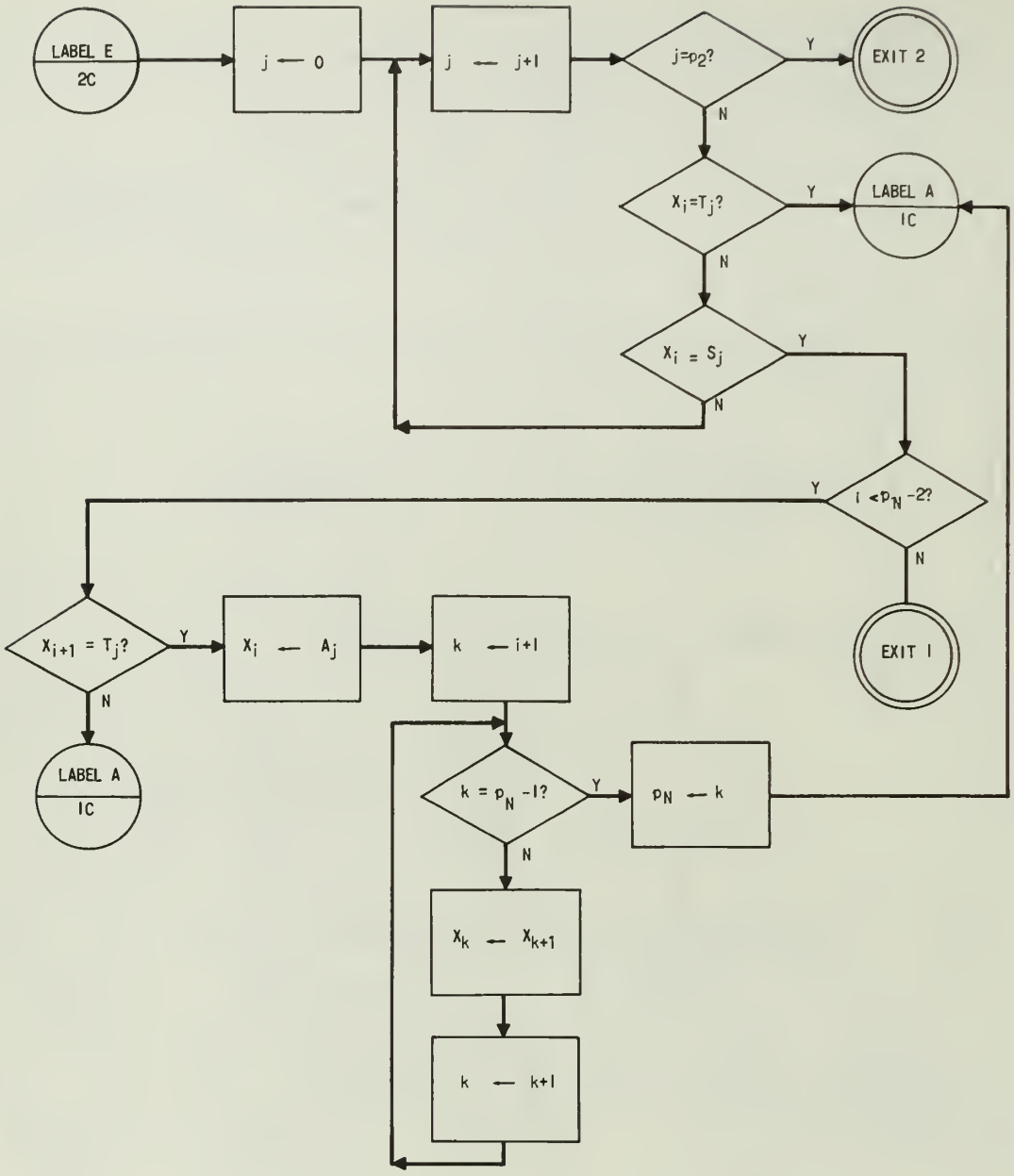


PROCEDURE: SCANNER [PAGE 1]

FIGURE 1C

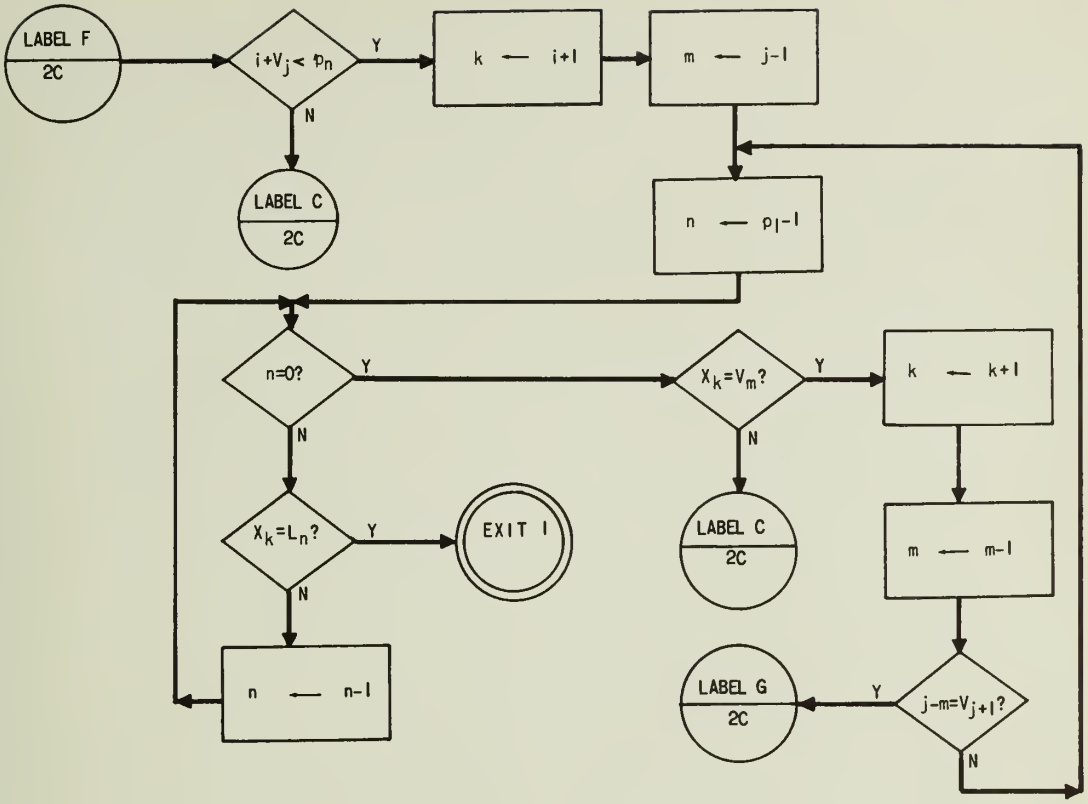


PROCEDURE: SCANNER [PAGE 2]
FIGURE 2C



PROCEDURE: SCANNER [PAGE 3]

FIGURE 3C



PROCEDURE: SCANNER [PAGE 4]
FIGURE 4C

NOV 10 1967

JUN 20 1989

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no. 237-242(1967
Parallelism exposure and exploitation in
3 0112 088398331